VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ENGINE V GLSL

GLSL BASED ENGINE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MICHAL ŠLESÁR**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ MILET, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Šlesár Michal**
Program: Informační technologie
Název: **Engine v GLSL**
GLSL Based Engine
Kategorie: Počítačová grafika

Zadání:

1. Nastudujte techniky GPGPU, OpenGL, Vulkan, GLSL, architektury grafických karet a tvorbu herních enginů.
2. Navrhněte engine, který poběží téměř celý na grafické kartě.
3. Implementujte navržený engine v jazyce GLSL a sadu příkladů využívající engine.
4. Proměřte a zhodnoťte výsledky.
5. Navrhněte pokračování a vytvořte demonstrační video.

Literatura:

- John Kessenich, et. al.: The OpenGL Shading Language, Version 4.60.7 (2019).
- Inigo Quilez and Pol Jeremias: Shadertoy (2021). https://www.shadertoy.com/.
- Marek Vinkler, et. al.: Massively Parallel Hierarchical Scene Processing with Applications in Rendering. Computer Graphics Forum (2013).
- Kshitij Gupta, et. al.: A Study of Persistent Threads Style GPU Programming for GPGPU Workloads (2012).

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2 a kostra aplikace.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Milet Tomáš, Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 3. května 2022

Abstrakt

Tvorba grafickej aplikácie spúštanej na GPU typicky obnáša konfiguráciu GPU, vytvorenie a konfiguráciu potrebných objektov a následne implementáciu samotného chovania aplikácie. Cieľom práce je za pomoci aplikačného rozhrania OpenGL vytvoriť nástroj, ktorý by túto konfiguráciu automatizoval. Užívateľ by vďaka tomu nemusel strácať čas konfiguráciou a mohol by rýchlo tvoriť a prototypovať grafické aplikácie. Vytvorený nástroj navyše aplikácii pridáva rôzne rozširujúce možnosti, ktoré nie sú natívne na GPU dostupné alebo podporované, ako napríklad práca s myšou a klávesnicou.

Abstract

Creating a graphical application running on a GPU typically involves configuring the GPU, creating and configuring the required objects, and then implementing the application's behavior itself. The aim of this work is to create a tool that would automate this configuration using the OpenGL application interface. As a result, the user would not have to waste time configuring and could quickly create and prototype graphics applications. In addition, the created tool adds new functionality to the application that is not native or supported on the GPU, such as working with a mouse and keyboard.

Kľúčové slová

grafická karta, GPU, GPGPU, CPU, OpenGL, Shadertoy, GLSL, C++, shader, program, pipeline, grafická pipeline, výpočtová pipeline, vertex, fragment, preprocesor, VAO, SSBO, buffer, textúra, framebuffer, uniforma, engine, regex, konfigurácia, automatizácia

Keywords

graphics card, GPU, GPGPU, CPU, OpenGL, Shadertoy, GLSL, C++, shader, program, pipeline, graphics pipeline, compute pipeline, vertex, fragment, preprocessor, VAO, SSBO, buffer, texture, framebuffer, uniform, engine, regex, configuration, automatisation

Citácia

ŠLESÁR, Michal. *Engine v GLSL*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Milet, Ph.D.

Engine v GLSL

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Tomáša Mileta, Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal informácie pre tvorbu tejto práce.

.....
Michal Šlesár
5. mája 2022

Poďakovanie

Rád by som sa poďakoval vedúcemu práce Ing. Tomášovi Miletovi, Ph.D. za jeho odbornú pomoc naprieč celou tvorbou tejto práce a za pôvodnú myšlienku, na ktorej bola táto práca postavená.

Obsah

Kapitola 1

Úvod

Grafická karta, respektíve GPU, sa už dnes nachádza takmer v každom zariadení. Zatiaľ čo centrálny procesor, respektíve CPU, má poskytovať všestrannú funkcionalitu pre implementáciu akéhokoľvek algoritmu, GPU je určená na akceleráciu špecifických výpočtov. Medzi tieto výpočty patrí napríklad grafické vykresľovanie.

Program spúšťaný na CPU si počas behu vytvára a konfiguruje zdroje podľa potreby, napríklad alokuje pamäť. Program spúšťaný na GPU je v tomto ohľade odlišný, nakoľko predpokladá že zdroje ktoré využíva, boli vytvorené a správne nakonfigurované už pred jeho spustením. Vytvorenie zdrojov a konfiguráciu GPU je nutné vykonávať z programu bežiaceho na CPU pomocou vybraného aplikačného rozhrania implementovaného ovládačmi konkrétnej GPU.

Pri jednoduchších aplikáciách z tohto vyplýva fakt, že užívateľ častokrát strávi viac času správnou konfiguráciou GPU, ako implementáciou samotného chovania aplikácie. Zároveň fakt, že zdanlivo jeden ucelený program je zložený z programov dvoch, jedného ktorý je spúšťaný na CPU a druhého ktorý je spúšťaný na GPU, môže spôsobovať rôzne problémy. Medzi tieto problémy môže patriť napríklad obtiažne ladenie.

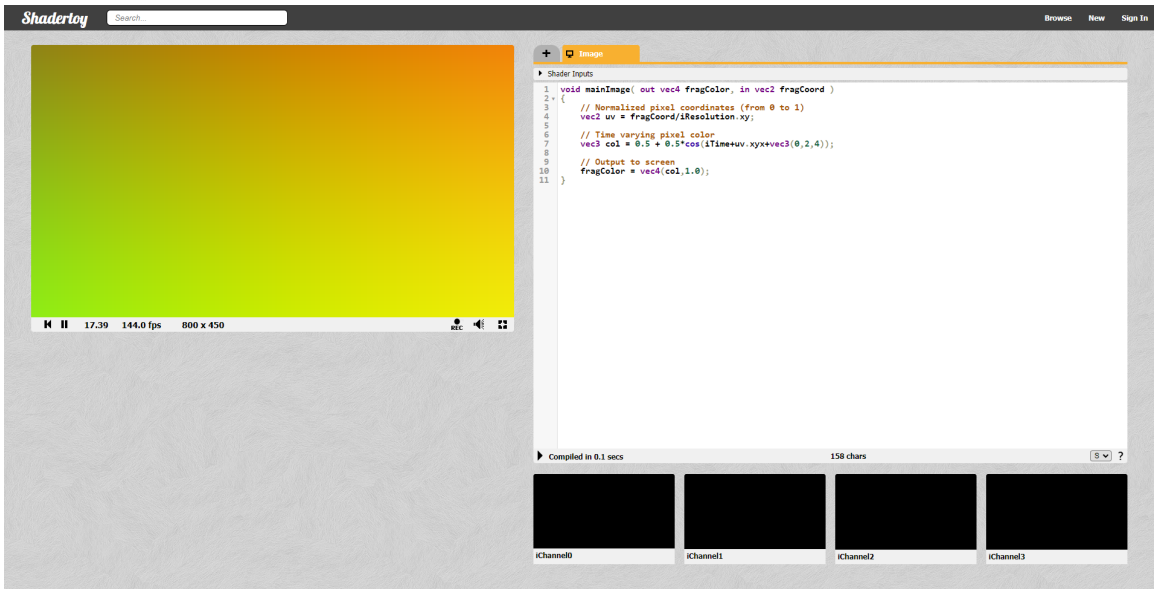
Preto je cieľom tejto práce vytvoriť nástroj s využitím OpenGL, pomocou ktorého je možné rýchlo a jednoducho tvoriť a prototypovať grafické aplikácie spúšťané na GPU. Tento nástroj na základe statickej analýzy vstupného kódu popisujúceho chovanie aplikácie, automaticky nakonfiguruje GPU a vytvorí zdroje potrebné pre jeho úspešné spustenie. Užívateľ teda bude môcť efektívne pracovať iba s GPU a vylúčiť prácu s CPU. Vzhľadom na tento fakt, poskytne nástroj užívateľovi navyše rozširujúcu funkcionalitu, ktorá nieje natívne na GPU dostupná, ako napríklad priama práca s myšou a klávesnicou, ktorá umožní tvorbu aj relatívne komplexnejších interaktívnych grafických aplikácií, ako napríklad hier.

Na úvod práce je v kapitole 2 zmienený už existujúci nástroj, ktorý čiastočne rieši zmienený problém. Nasleduje kapitola 3, ktorá obsahuje teoretický základ potrebný pre navrhnutie a vytvorenie cieľového nástroja, teda ako funguje samotná GPU a popis aplikačného rozhrania OpenGL. Nasledujúca kapitola 4 popisuje návrh a kapitola 5 popisuje implementáciu cieľového nástroja. Cieľom kapitoly 6 je porovnať a zhodnotiť dosiahnuté výsledky.

Kapitola 2

Stav existujúcich riešení

V súčastnosti už existuje nástroj zvaný Shadertoy[1], ktorý čiastočne pokrýva cieľ tejto práce. Tento nástroj umožňuje priamo vo webovom prehliadači jednoducho vytvárať grafické aplikácie bežiace na GPU a v rámci prívetivého užívateľského rozhrania zobraziť náhľad spustenej aplikácie, viď obrázok 2.1. Nástroj ďalej umožňuje definovať rôzne vstupy, ktoré je možné v kóde programu využiť, ako napríklad textúry, vstup klávesnice, myši, webkamery apod. Vytvorené programy je možné zdieľať a taktiež je možné prehliadať programy vytvorené inými užívateľmi.



Obr. 2.1: Užívateľské rozhranie nástroja Shadertoy. V ľavej časti je možné vidieť náhľad spustenej aplikácie. V pravej časti sa nachádza editor kódu a tlačidlá pre špecifikáciu vstupov.

Problémom tohto nástroja je technická limitácia aplikačného rozhrania WebGL, ktoré je využité pre prácu s GPU. V porovnaní s ostatnými štandardnými aplikačnými rozhraniami, ako napríklad OpenGL, DirectX alebo Vulkan, disponuje WebGL pomerne malou a základnou funkcionalitou. Z tohto dôvodu je možné v tomto nástroji pracovať iba s fragmentovým shaderom, ktorý je určený pre spracovanie pixelov. Jeho použitie teda spočíva

[1]Nástroj Shadertoy je možné vyskúšať priamo v prehliadači na adrese: https://shadertoy.com

prevažne vo vytváraní a zdieľaní grafických efektov, nakoľko nie je možné pracovať s radou ďalších typov shaderov, ktoré by rozšírili potencionálne využitie na komplexnejšie aplikácie.

Cieľom tejto práce teda nie je vytvoriť klon nástroja Shadertoy s podobným užívateľským rozhraním, ale zamerať sa primárne na vyššie uvedený problém a vytvoriť flexibilný nástroj, ktorý by užívateľovi umožnil využiť plnú silu a možnosti GPU.

Kapitola 3

Teória

Táto kapitola popisuje teoretický základ potrebný pre pochopenie súvislostí, za účelom vytvorenia popísaného nástroja. Keďže úlohou tohto nástroja je vytvárať grafické aplikácie na GPU, je nutné ukázať čo to vlastne GPU je, ako funguje a ako sa s ňou pracuje, viď sekcia 3.1. Druhá sekcia 3.2 následne podrobne popisuje technológiu OpenGL, pomocou ktorej bola táto práca vytvorená.

3.1 Grafická karta

Grafická karta, alebo GPU (Graphics Processing Unit), má radu využití. Typicky sa jedná o grafické vykresľovanie, no v posledných rokoch sa do popredia dostávajú rôzne iné využitia, ako napríklad strojové učenie a spracovanie videí. GPU ktorá sa využíva na všeobecné výpočty, ktoré nutne nesúvisia s grafickým vykresľovaním sa preto zvykne označovať ako GPGPU (General Purpose Graphics Processing Unit). V tejto sekcii je najprv vysvetlený dôvod vzniku GPU pomocou porovnania jej návrhu voči CPU, viď podsekcia 3.1.1. V podsekcii 3.1.2 je následne popísaná architektúra GPU a v záverečnej podsekcii 3.1.3 je popísaný spôsob, akým sa s GPU pracuje a komunikuje na softvérovej úrovni. Fakty a poznatky využité pri tvorbe tejto sekcie boli čerpané zo sprievodných materiálov ku viacerým seminárom o architektúre GPU [7, 1, 6].

3.1.1 Dôvod vzniku GPU

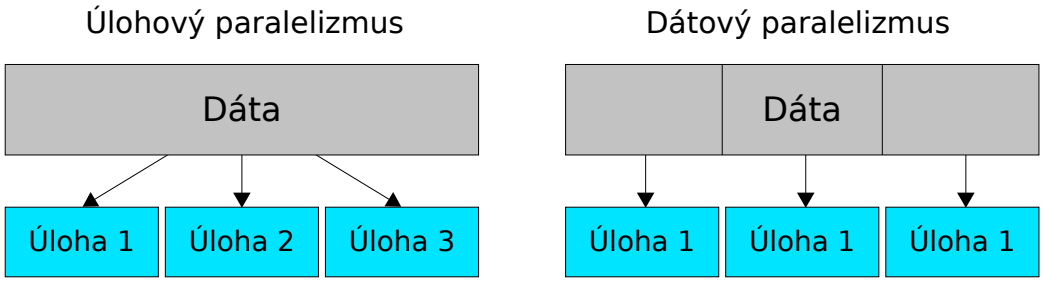
GPU bola vyvinutá za účelom zrýchlenia výpočtov, na ktoré je CPU príliš pomalé. V dobe vzniku GPU sa jednalo primárne o akceleráciu 2D a neskôr 3D grafických operácií, nakoľko vtedajšie CPU značne limitovali potencionálny technologický posun v tejto oblasti. Pre pochopenie dôvodu, prečo tomu tak je, je potrebné definovať pojmy latencia a priepustnosť a následne sa pozrieť na to, ako funguje CPU.

- **Latencia** – časové oneskorenie v pozorovanom systéme medzi akciou a reakciou na túto akciu. Napríklad medzi momentom odchodu autobusu z autobusovej stanice a jeho príchodom do cieľovej zastávky. Možné znížiť napríklad rýchlejším autobusom.
- **Priepustnosť** – množstvo práce vykonané za jednotku času. Napríklad počet prenesených pasažierov z autobusovej stanice do cieľovej zastávky. Možné zvýšiť napríklad vyšším počtom autobusov.

Ako už z názvu plynie, CPU je centrálna jednotka. Keďže je centrálna, v princípe nemá špecifické využitie ako GPU a teda spracováva vysoký počet rôznorodých a z pohľadu CPU typovo neznámych úloh. Úlohy zároveň súperia o výpočtový čas, ktorý im je prideľovaný plánovačom operačného systému, takže je vhodné aby úloha skončila čo najskôr. Pri týchto úlohách je teda cieľom zaručiť čo najmenšiu latenciu, aby boli prichádzajúce úlohy čo najrýchlejšie spracované a čakajúce úlohy neboli príliš dlho blokované. Veľký problém z pohľadu návrhu CPU je rýchlosť prístupu k pamäti RAM (Random Access Memory). Keďže RAM sa fyzicky nenachádza v CPU, trvá pomerne dlho z nej načítať dáta a teda sa zvyšuje latencia. Tento problém je minimalizovaný prítomnosťou vyrovnávajúcej pamäti v CPU, takzvanou CPU cache, ktorá dočasne ukladá časť aktuálne využívaných dát alebo dáta pri ktorých je predpoklad, že budú v blízkej dobe použité. Prítomnosť CPU cache a jej veľkosť teda znižuje počet prístupov do pamäte RAM a redukuje latenciu. Priorita CPU je síce latencia, ale stále je nutné klásť dôraz aj na priepustnosť. CPU sa teda skladá z viacerých samostatných jednotiek, zvaných jadro, ktoré pracujú nezávisle od seba a teda umožňujú paralelné spustenie viacerých úloh.

Paralelizmus je na základe problému úlohy možné kategorizovať dvoma možnými spôsobmi (pre grafické znázornenie rozdielu viď obrázok 3.1):

- **Úlohový paralelizmus** – Rozloženie viacero odlišných úloh na dostupné jadrá. Cieľ je čo najrýchlejšie spracovať jednotlivé úlohy (CPU). Čím nižšia latencia, tým lepšie, aj na úkor nižšej priepustnosti.
- **Dátový paralelizmus** – Paralelné spustenie jednej konkrétnej úlohy na viacerých jadrách tak, že každé jadro vykonáva tú istú úlohu a pracuje s inou časťou vstupných dát. Cieľ je čo najrýchlejšie spracovať všetky dáta. Čím vyššia priepustnosť, tým lepšie, aj na úkor vyššej celkovej latencie.



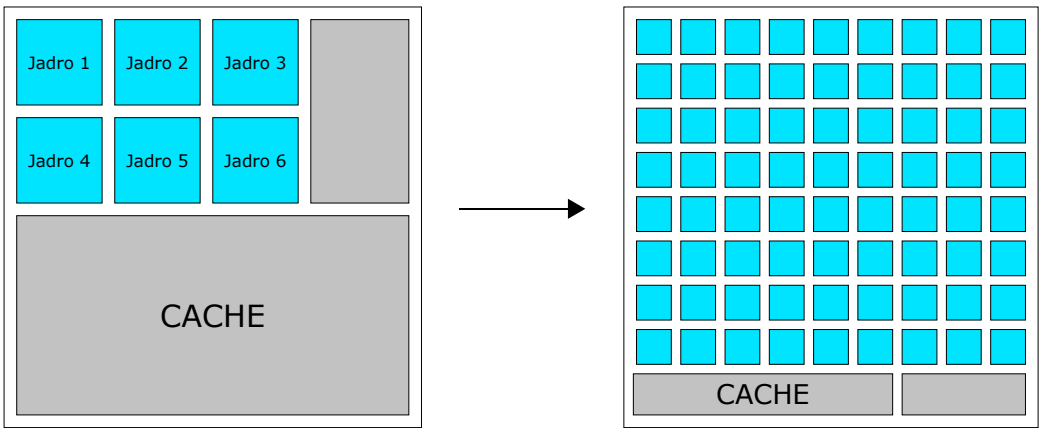
Obr. 3.1: Grafická vizualizácia rozdielu úlohového a dátového paralelizmu.

Na základe vyššie popísaných faktov môže vzniknúť otázka, prečo nemôže mať CPU jednoducho viac jadier a vďaka tomu nie len nízku latenciu, ale aj vysokú priepustnosť.

Všetky výpočtové jednotky sa na fyzickej úrovni skladajú z tranzistorov. Týchto tranzistorov je do daného čipu možné zmestiť iba obmedzený počet. Keďže priorita CPU je úlohový paralelizmus, jednotlivé jadrá musia disponovať rôznymi mechanizmami a taktiež disponovať značne veľkou, už spomínanou, CPU cache, ktorá je taktiež zložená z tranzistorov. Potencionálny počet tranzistorov ktoré je teda možné priradiť samotnému výkonu jadier, je typicky len malý zlomok z celkového počtu. Pridanie ďalších jadier pri tom istom celkovom počte tranzistorov by teda znamenalo zredukovať potencionálny výpočtový výkon ostatných jadier, prípadne znížiť veľkosť cache, čo by ale malo dopad na výslednú latenciu.

Cieľom CPU je teda úlohový paralelizmus a pre zaručenie efektívneho úlohového paralelizmu je potrebná čo najnižšia latencia. Potencionálna efektivita úloh vyžadujúcich dátový paralelizmus je z tohto dôvodu obmedzená. Príklad takejto úlohy je napríklad vykresľovanie, kde sa vykonáva ten istý algoritmus nad potencionálne obrovským počtom objektov tvoriacich výsledný obraz. Vysoký počet samostatných výpočtových jednotiek je pre tento prípad ideálny, nakoľko každá výpočtová jednotka môže spracovať iný objekt a v konečnom dôsledku bude výsledný obraz spracovaný podstatne rýchlejšie.

Pre riešenie tohto problému boli vyvinuté špecializované výpočtové jednotky nazývané GPU, navrhnuté a optimalizované pre akceleráciu výpočtov využívajúcich dátový paralelizmus. Vysoká priepustnosť je zaručená značne vyšším počtom jadier, respektíve samostatných výpočtových jednotiek. Keďže hlavná priorita GPU je priepustnosť a nie latencia, potreba cache a iných mechanizmov pre redukciu latencie je nižšia. Tranzistory, ktoré by normálne boli použité pre cache a tieto mechanizmy, je vďaka tomu možné využiť pre nové jadrá. Nižšia je taktiež potreba na výkon jednotlivých jadier, čo uvoľní miesto pre ďalšie nové jadrá. Túto zmenu názorne ilustruje obrázok 3.2. Grafická karta obsahujúca GPU čip navyše typicky fyzicky disponuje vlastnou vysokorýchlostnou pamäťou VRAM (Video RAM) a nepracuje s pamäťou RAM. Prístup k tejto pamäti je tým pádom značne rýchlejší a efektívnejší.



Obr. 3.2: Názorná ukážka rozdielneho využitia tranzistorov pri GPU. Značný počet tranzistorov sa uvoľnil vďaka možnosti zmenšiť cache. Ďalšie tranzistory sa uvoľnili vďaka nižšej náročnosti na výkon a mechanizmy jednotlivých jadier a teda vyžadujú menší počet tranzistorov. Tieto tranzistory je možné využiť pre pridanie nových jadier za účelom zvýšenia potencionálnej priepustnosti.

Príklad

Pomyslená 3D scéna sa skladá z 1000 objektov. Pomyslené CPU disponuje 6 jadrami a jedno jadro spracuje jeden objekt priemerne za 1 jednotku času. Pomyslené GPU disponuje 208 jadrami a jedno jadro spracuje jeden objekt priemerne za 30 jednotiek času (zvolené čísla majú za úlohu demonštrovať vplyv prioritizácie priepustnosti a latencie). Pre zachovanie jednoduchosti je časová réžia plánovania a podobných mechanizmov zanedbaná. Úlohou je vypočítať celkovú dobu spracovania danej 3D scény.

$$\begin{aligned} (1000.1)/6 &= 166.67j \\ (1000.30)/208 &= 144.23j \end{aligned} \tag{3.1}$$

Z výpočtu 3.1 vyplýva, že aj napriek tomu, že jednotlivé jadrá GPU sú niekoľko násobne pomalšie, výsledná doba spracovania celej 3D scény je oproti CPU nižšia. Tento rozdiel by sa navyše priamo úmerne zvyšoval s vyšším počtom objektov v prospech GPU. Tento príklad síce neodzrkadľuje reálny rozdiel, ale názornou ukážkou demonštruje výhodu vysokej priepustnosti pri úlohe vyžadujúcej dátový paralelizmus.

3.1.2 Architektúra GPU

Terminológia architektúry GPU je relatívne problematická. Rôzni výrobcovia označujú rôzne komponenty a mechanizmy určené na zdanlivo rovnakú vec iným výrazom. Samotná architektúra GPU je taktiež odlišná medzi rôznymi výrobcami a rôznymi generáciami GPU. Z tohto dôvodu je nasledujúci teoretický základ a terminológia inšpirovaná modernými GPU výrobcu NVIDIA. Pre potreby tejto práce navyše nieje potrebné podrobne rozoberať všetky jednotlivé komponenty GPU a spôsob akým fungujú. Architektúra je teda popísaná jednoduchým spôsobom, ktorý je plne dostačujúci pre pochopenie potrebných súvislostí.

GPU je na základe typu možné kategorizovať na integrovanú a dedikovanú (pre grafické znázornenie rozdielu viď obrázok 3.3).

- **Integrovaná** – CPU a GPU sa fyzicky nachádza v tom istom čipe (CPU čip). GPU nemá dedikovanú pamäť a pre ukladanie dát využíva RAM. Nižší výkon vzhľadom na obmedzený počet tranzistorov a relatívne pomalý prístup k pamäti RAM.
- **Dedikovaná** – GPU sa fyzicky nachádza na osobitnom čipe. Disponuje dedikovanou vysokorýchlostnou pamäťou iba pre účely GPU (VRAM). Vyžaduje vlastné chladenie a napájanie. Oveľa vyšší potencionálny výkon v porovnaní s integrovanou GPU.

Obr. 3.3: Grafická vizualizácia rozdielu integrovanej a dedikovanej GPU. Zatiaľ čo integrovaná GPU sa nachádza fyzicky na CPU čipe, dedikovaná GPU má osobitný čip a vlastnú vysokorýchlostnú pamäť.

GPU sa skladá z viacerých samostatných fyzických jednotiek, ktoré sa označujú ako *Streaming Multiprocessor* (SM). SM sa ďalej delí na ďalšie výpočtové jednotky, označované ako *Streaming Processor* (SP). Akcia vykonávaná na jednom SP v konkrétnom SM, je nezávislá od inej akcie, vykonávanej na inom SP. SP sa na záver delia na samostatné virtuálne jednotky, *vlákna*. Počet vlákien jednotlivých SP nie je pevne daný, ale závisí od programu, ktorý je na SP spúšťaný, respektíve od náročnosti tohto programu na zdroje. SM obsahujú vysoký počet 32-bitových registrov (žiadne ukladanie a obnovovanie stavu registrov ako pri CPU, dáta sú perzistentné počas celého výpočtu), ktoré sú rýchlo priraďované medzi jednotlivé SP podľa potreby. Plánovanie úloh na GPU vykonáva na globálnej úrovni *globálny plánovač* a na úrovni jednotlivých SM *plánovač warpov*. Medzi základný typ vyrovnávacích pamätí (cache), ktoré GPU obsahuje, patrí napríklad L2 cache (globálna cache

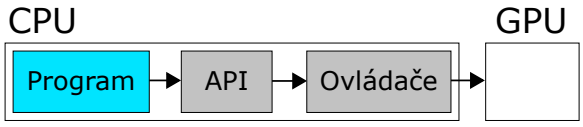
ku ktorej majú prístup všetky SM) a L1 cache (pamäť SM ku ktorej majú prístup všetky SP daného SM). Tieto cache sú typicky v porovnaní s CPU veľmi malé. Hardvér SM je navrhnutý na základe princípu SIMD (Single Instruction Multiple Data), ktorý je využívaný pre dátový paralelizmus.

Spustenie úlohy

Spúšťaná úloha na GPU je definovaná sekvenciou inštrukcií a požadovaným počtom vlákien. Tento počet vlákien predstavuje *thread block*, ktorý je spúšťaný na GPU. Thread block je následne rozdelený po 32 vláknach do celkov nazývaných *warp* (pri AMD GPU *wavefront* zložený z 64 vlákien). Pokiaľ nie je celkový počet vlákien násobkom čísla 32, niektoré vlákna v rámci posledného warpu budú neaktívne. Warpy využívajú princíp SIMT (Single Instruction Multiple Threads), kde je tá istá inštrukcia spustená na viacerých vláknach. Každé ztýchto vlákien pracuje s inou časťou dát. Jednotlivé vlákna tvoriace warp sú síce spúšťané sériovo, ale samotné warpy paralelne. Rozdielne warpy môžu spúšťať inú časť kódu. Globálny plánovač môže spustiť thread block na konkrétnom SM iba v prípade, že toto SM má k dispozícii zdroje pre jeho spustenie. Úloha, respektíve thread block, je dokončený vtedy, keď sú dokončené všetky warpy ktoré ho tvoria. Warpy spúšťa (mapuje na SP), plánuje a nastavuje plánovač warpov.

3.1.3 Spôsob komunikácie s GPU

Program spúšťaný na CPU komunikuje s nejakou perifériou alebo iným externým komponentom zariadenia pomocou rozhrania, ktoré je implementované ovládačmi daného zariadenia, viď obrázok 3.4. Toto rozhranie typicky navrhuje aj implementuje priamo výrobca daného komponentu.



Obr. 3.4: Ukážka spôsobu komunikácie programu bežiaceho na CPU s GPU. API obsahuje deklarácie funkcií pomocou ktorých je možné s GPU pracovať. Ovládače musia obsahovať implementáciu týchto funkcií. Program túto implementáciu využije a môže úspešne komunikovať s GPU.

Architektúry GPU sa podstatne líšia naprieč rôznymi výrobcami a generáciami. Aby bol softvér pracujúci s GPU prenositeľný, musí pri GPU existovať štandardné aplikačné rozhranie, ďalej len API, pomocou ktorého by bolo možné komunikovať s každou GPU rovnakým spôsobom, bez ohľadu na to ako reálne daná GPU na pozadí funguje. Tieto API sú teda typicky navrhované rôznymi organizáciami, ktoré priamo nenavrhujú a nevyrábajú GPU (za výnimku je možné považovať niektoré proprietárne technológie niektorých výrobcov). Pokiaľ chce následne výrobca GPU toto API podporovať, musí ho implementovať a túto implementáciu dodať v rámci ovládačov danej GPU. Vývoj a technologický pokrok v oblasti GPU je veľmi rýchly. Organizácie, ktoré GPU API navrhujú, teda musia neustále na tieto zmeny reagovať a odzrkadľovať ich vo forme novej verzie.

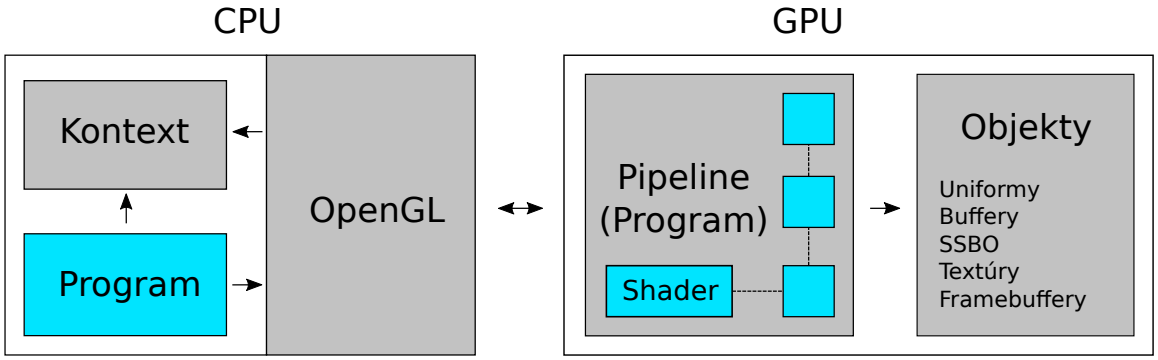
Medzi štandardné GPU API patrí:

- **OpenCL** (Open Computing Language) – Umožňuje tvorbu výpočtových programov (GPGPU). Zaujímavé na tejto API je to, že jej implementácia je dostupná aj na CPU, FPGA čipoch a rôznych hardvérových akcelerátoroch, nie iba na GPU.
- **CUDA** (Compute Unified Device Architecture) – Podobne ako OpenCL umožňuje tvorbu výpočtových programov. Proprietárna technológia vyvíjaná firmou NVIDIA a podporovaná iba na ich vlastných GPU.
- **OpenGL** (Open Graphics Library) – Určené pre vykresľovanie 3D grafiky. Prvotné verzie značne obmedzujúce z dôvodu vysokého počtu abstrakcií. Najnovšia verzia umožňuje tvorbu všeobecných výpočtových programov. Podporované na väčšine moderných platforiem.
- **Vulkan** – API poslednej generácie. Obsahuje minimum abstrakcií oproti OpenGL za účelom minimalizácie réžie. Väčšinu technických problémov, ktoré riešilo OpenGL za programátora, musí vo Vulkane vyriešiť programátor sám. Práca s týmto API je teda náročnejšia, no ponúka väčšiu flexibilitu. Podporované, podobne ako OpenGL, na väčšine moderných platforiem.
- **DirectX** – Alternatíva OpenGL a Vulkanu pre vykresľovanie 3D grafiky. Zatiaľ čo staršie verzie fungovali na podobnom princípe ako OpenGL, najnovšia verzia tejto API, DirectX 12, funguje na podobnom princípe ako Vulkan. Proprietárna technológia pre systém Windows a hernú konzolu Xbox.
- **Metal** – Proprietárna technológia firmy Apple. Najnovšie zariadenia firmy Apple podporujú iba Metal a staršie verzie OpenGL a OpenCL.

3.2 OpenGL

OpenGL (Open Graphics Library) je API vyvíjané konzorciom Khronos Group, pre komunikáciu s GPU za účelom vykresľovania 2D a 3D grafiky. Na rozdiel od alternatívneho aplikačného rozhrania DirectX, vyvíjaného firmou Microsoft, je toto rozhranie podporované na viacerých populárnych platformách, ako napríklad Windows, Linux, Mac OS X, ale aj Android ktorý využíva špeciálnu verziu OpenGL ES. Je definovaná ako zoznam funkcií a konštánt, ktoré môže aplikácia využiť. Je nutné podotknúť, že OpenGL je iba API, nie knižnica, viď obrázok 3.1.3. Teda to, ako sú jednotlivé funkcie popísané týmto rozhraním implementované, z veľkej časti závisí od výrobcu GPU, ktorý toto API za účelom podpory musí implementovať. Z tohto dôvodu sa nemusí OpenGL program spúšťaný na GPU jedného výrobcu správať ako ten istý program spúšťaný napríklad na GPU iného výrobcu. Tieto odchýlky správania sú ale typicky minimálne.

Na obrázku 3.5 je možné vidieť jednoduchú schému práce s GPU pomocou OpenGL. Prvá podsekcia 3.2.1 podrobnejšie popisuje jednotlivé pojmy na tejto schéme a zároveň popíše niektoré zo základných typov objektov. Nasledujúca podsekcia 3.2.2 popisuje jazyk GLSL pre programovanie shaderov. Podsekcie 3.2.3 a 3.2.4 popisujú programovateľné procesy (pipeline) OpenGL. Fakty a poznatky využité pri tvorbe tejto sekcie boli čerpané z oficiálnej špecifikácie OpenGL verzie 4.6 [8], oficiálnej špecifikácie jazyka GLSL verzie 4.60.7 [2] a kníh pre pokročilú prácu s OpenGL [5, 10]. Keďže sa jedná o komplexnú technológiu, preberané mechanizmy sú vysvetlené čo najjednoduchšie pre potreby tejto práce.



Obr. 3.5: Znázornenie akým spôsobom funguje OpenGL. Program bežiaci na CPU využíva OpenGL API pre komunikáciu s GPU. Aktuálny stav je uložený v kontexte OpenGL. Interné procesy GPU (pipeline) je možné programovať pomocou shaderov. Jednotlivé shadery počas behu pristupujú k objektom, ktorých dáta sú typicky uložené vo VRAM a ich stav v kontexte.

3.2.1 Základné pojmy

Jednotlivé základné pojmy v tejto sekcii sú zoradené hierarchicky, od najzákladnejšieho po pokročilejšie. Každý jeden pojem nejakým spôsobom nadväzuje na niektorý z predchádzajúcich. Na konci popisu každého pojmu je možné nájsť stranu (rozsah strán) oficiálnej špecifikácie OpenGL, na ktorej sa nachádza jeho rozsiahlejšie a podrobnejšie vysvetlenie.

Kontext

Za účelom využívania OpenGL API je nutné v programe bežiacom na CPU vytvoriť inštanciu OpenGL, zvanú kontext a aktivovať ju (v jednom vlákne CPU môže byť aktívny iba jeden OpenGL kontext naraz). V kontexte sú uložené všetky nastavenia danej inštancie OpenGL. Je možné vytvoriť viacero kontextov s odlišnými nastaveniami a prepínať medzi nimi podľa potreby. Keďže vytváranie kontextu závisí od konkrétneho operačného systému (platformy), OpenGL tieto operácie nepokrýva. Samotné funkcie OpenGL API implicitne využívajú práve aktívny kontext. Prerekvizita ich spustenia je teda správne vytvorený a aktivovaný kontext. [8, s. 2]

Objekt

OpenGL objekt je konštrukcia, ktorá obsahuje nejaký stav. Tieto objekty je možné vytvárať, konfigurovať a mazať pomocou príslušných funkcií OpenGL. Vytvorený objekt a jeho aktuálny stav je uložený v rámci OpenGL kontextu. Každý objekt má priradený unikátny názov (ID), ktorý jednoznačne identifikuje daný objekt. Názvy objektov sú reprezentované nezáporným číslom. [8, s. 27–29]

Program

Niektoré špecifické časti interných procesov OpenGL sú programovateľné nezávislými programami zvanými *shader* [8, s. 29]. OpenGL program je objekt, ktorý mapuje typ (časť procesu) na konkrétny shader. Pri spustení niektorého z takýchto procesov sa následne v programovateľných častiach implicitne využijú shadery aktívneho programu. V rovna-

kom čase môže byť aktívny najviac jeden program. Vstavané vstupy a výstupy jednotlivých shaderov závisia od jeho typu. [8, s. 29–30]

Uniformná premenná

Jednotlivé shadery môžu obsahovať pomenované uniformné premenné podporovaného typu. Uniformné premenné shaderov zdieľajú jeden globálny menný priestor v rámci OpenGL programu, ktorého sú súčasťou. Pomocou uniformných premenných môže aplikácia bežiaca na CPU poslať OpenGL programu dáta (jeho shaderom). [2, s. 132–140]

Buffer

Buffer je OpenGL objekt, ktorého úloha je spravovať konkrétny blok pamäte (sekvencia bytov). Informácie o tomto bloku sú uložené v stave daného objektu (dáta tohto bloku sa nenachádzajú v jeho stave). Pomocou funkcií OpenGL je možné buffer vytvoriť, modifikovať jeho stav a pracovať s jeho dátami. OpenGL umožňuje naviazať buffer na niektorý z podporovaných typov. Na každý typ môže byť naviazaný v rovnakom čase najviac jeden buffer (väzby sú uložené v kontexte). Niektoré funkcie využívajú tieto väzby a implicitne pracujú s bufferom, ktorý je naviazaný na daný typ. [8, s. 1–62]

SSBO

SSBO (Shader Storage Buffer Object) je špeciálny typ bufferu, pomocou ktorého môžeme pracovať s nejakým objektom bufferu (čítanie a zapisovanie obsahu daného bufferu) priamo zo shaderu OpenGL programu. SSBO teda môže byť využitý napríklad ako spôsob komunikácie dvoch rozdielnych inštancií shaderov alebo programov. [8, s. 151–153]

Textúra

Textúra je OpenGL objekt, ktorý typicky reprezentuje obraz. Priestor textúry môže mať jednu až tri dimenzie a jednotlivé prvky tohto priestoru sa nazývajú *texel* (pixel). Hlavné využitie textúr je nanášanie detailov na časti vykresľovaných objektov. Pomocou funkcií OpenGL je možné textúru vytvoriť, modifikovať jej stav (konfiguráciu) a pracovať s jej dátami. Pre prístup k objektu textúry v rámci shaderu je nutné využiť uniformnú premennú typu *sampler* alebo *image*. Základný rozdiel medzi týmito dvoma typmi relevantný pre túto prácu je, že pomocou typu sampler je možné z textúry informácie iba čítať a pomocou typu image je možné do nich priamo aj zapisovať. [8, s. 30]

Framebuffer

Framebuffer je OpenGL objekt, ktorý môže byť použitý ako cieľ vykresľovania. Framebuffer obsahuje viacero typov bufferov. Základný buffer framebufferu je color buffer, ktorého úlohou je napríklad uchovávať informáciu o farbe jednotlivých pixelov výstupného obrazu. Existuje predvolený framebuffer (framebuffer s ID 0), ktorý typicky predstavuje obrazovku (okno) zariadenia a jeho konfigurácia (stav) je závislá od platformy na ktorej je aplikácia spúšťaná. Predvolený framebuffer je vytvorený a nakonfigurovaný automaticky v rámci vytvárania OpenGL kontextu. Pomocou OpenGL je možné vytvoriť a konfigurovať nové framebuffery, ktoré môžu byť aktivované namiesto toho predvoleného. Na color buffer môžu byť naviazané textúry, vďaka čomu môže byť uložený výsledný obraz v danej textúre. V rovnakom čase môže byť aktívny práve jeden framebuffer. [8, s. 299–301]

3.2.2 GLSL

GLSL (OpenGL Shading Language) [2] je programovací jazyk pre písanie shaderov. Syntax jazyka GLSL vychádza z rodiny jazyka C, od ktorého zároveň preberá rôzne typické jazykové konštrukcie, ako napríklad *štruktúry*, *funkcie* a *direktívy preprocesoru*. Štandardná knižnica obsahuje rozličné matematické funkcie. Jednotlivé shadery sú nahrávané na GPU z programu bežiaceho na CPU v textovom formáte v jazyku GLSL, na ktorej sú následne skompilované do natívneho strojového kódu, ktorý je grafická karta schopná spustiť. Samotný preklad je vykonávaný prekladačom dodávaným výrobcom grafického čipu.

Výpis 3.1: Ukážka štruktúry kódu shaderu v GLSL. Každý shader začína deklaráciou verzie GLSL a skladá sa zo zoznamu vstupných a výstupných premenných (vstavané alebo definované užívateľom), uniformných premenných a funkcie main (vstupný bod shaderu). Zatiaľ čo vstupné a výstupné premenné spracováva OpenGL automaticky, v prípade že chce užívateľ použiť deklarovanú uniformnú premennú alebo SSBO, musí odpovedajúci objekt naviazať na túto deklaráciu pomocou príslušných funkcií OpenGL.

```
#version verzia_glsl

in type vstupnaPremenna1;
in type vstupnaPremanna2;

out type vystupnaPremenna1;

uniform type uniformnaPremenna1;

buffer ssbo1 {
    type premenna1;
    type premenna2;
};

void main()
{
    vystupnaPremenna1 = uniformna_premenna1;
}
```

Pri niektorých deklaráciách premenných v GLSL je možné špecifikovať rôzne parametre, ktoré konfigurujú danú premennú v rámci shaderu respektíve jeho OpenGL programu nasledujúcim spôsobom:

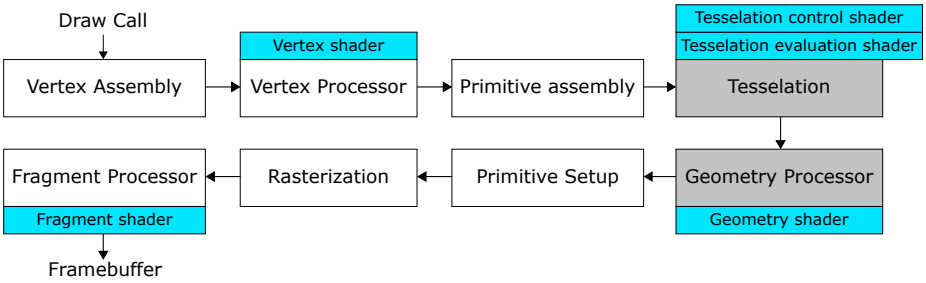
```
layout(parameter1, parameter2 = hodnota, ...) deklaracia;
```

Jazyk GLSL podporuje nasledujúce základné dátové typy ktoré je možné využiť v rámci takmer všetkých konštrukcií:

- **Skalárne typy** – bool, int, uint, float a double.
- **Vektor** - Môže byť zložený z dvoch až štyroch hodnôt rovnakého skalárneho typu.
- **Matica** - Matica môže nadobúdať hodnoty skalárneho typu float alebo double.

3.2.3 Grafická Pipeline

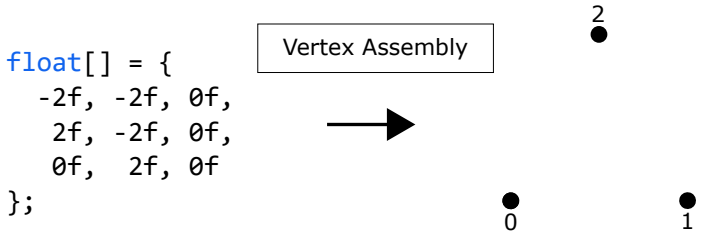
Grafická pipeline popisuje programovateľný proces vykresľovania. Spustenie procesu vykresľovania je možné pomocou viacerých funkcií OpenGL a zvykne sa označovať ako *draw call*. Celý proces je v jednoduchosti znázornený na obrázku 3.6 a cieľom tejto podsekcie je podrobnejšie popísať jeho jednotlivé časti.



Obr. 3.6: Znázornenie grafickej pipeline popisujúcej proces vykresľovania. Správanie niektorých častí je možné pozmeniť pomocou shaderu daného typu (modré bloky). OpenGL implicitne využíva shadery práve aktívneho OpenGL programu. Niektoré časti sú vynechateľné (šedé bloky). OpenGL takéto časti implicitne vynechá v prípade, že práve aktívny OpenGL program neobsahuje shader typu využívaného v tejto časti. Časti, ktorých správanie nie je možné pozmeniť shaderom, je možné prispôsobiť pomocou konfigurácie stavu kontextu príslušnými OpenGL funkciami.

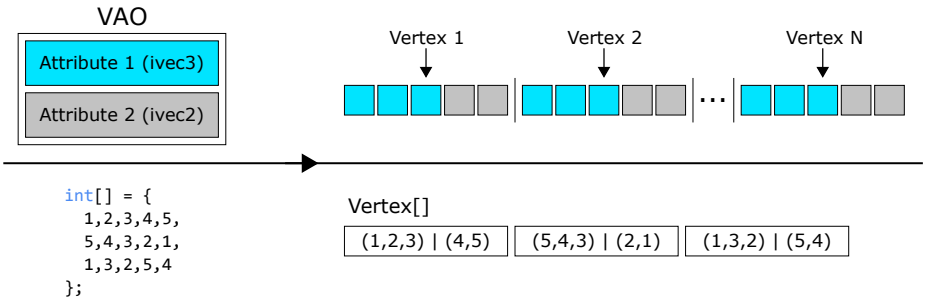
Vertex Assembly (Vstup)

Na vstupe grafickej pipeline sa nachádza *Vertex Assembly*. Vertex (vrchol) je kolekcia jedného alebo viacerých atribútov [8, s. 337]. Každý atribút môže obsahovať jednu až štyri skalárne hodnoty. Úlohou tejto časti je zostaviť vertexy zo vstupných dát, ktoré budú odoslané do ďalšej časti, viď obrázok 3.7.



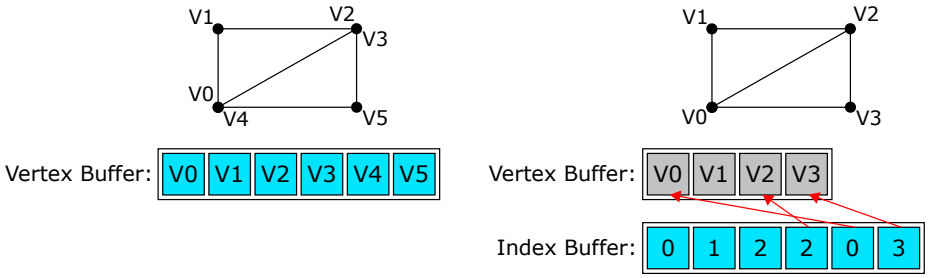
Obr. 3.7: Znázornenie spracovania dát za účelom zostavenia vertexov. Na vstupe sa nachádza OpenGL buffer, ktorý obsahuje hodnoty reprezentujúce pozíciu vertexu v priestore. Pozícia je v tomto prípade zložená z troch skalárnych hodnôt typu float.

Pre konfiguráciu očakávaného formátu jednotlivých atribútov sa využíva špeciálny OpenGL objekt: *Vertex Array Object* (VAO). VAO obsahuje zoznam vertex atribútov. Pre každý atribút VAO je potrebné špecifikovať, z akého typu skalárnych hodnôt sa skladá a koľko ich je. Na každý atribút je taktiež potrebné naviazať OpenGL buffer (vertex buffer), z ktorého budú pre daný atribút čerpané dáta. Atribút by ešte mohla zaujímať medzera v pamäti medzi nasledujúcimi vertex atribútmi (stride), prípadne kde v pamäti daný atribút začína (offset). Využívané je implicitne aktuálne aktívny VAO. Interpretácia VAO je znázornená na obrázku 3.8.



Obr. 3.8: Názorná ukážka interpretácie VAO. VAO má na tomto obrázku zapnuté dva atribúty. Prvý má priradený typ ivec3 (tri skalárne hodnoty typu int) a druhý má priradený typ ivec2 (dve skalárne hodnoty typu int). Vstupné dáta znázornené na obrázku by teda boli na základe tohto VAO interpretované ako tri vertexy.

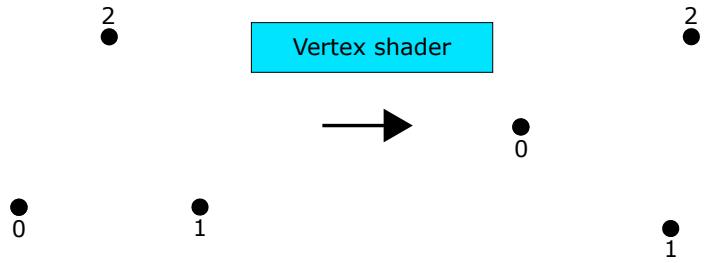
Pre každý VAO atribút je možné, v prípade potreby, zapnúť „indexované“ načítanie dát pomocou *index bufferu*. Index buffer je, podobne ako vertex buffer, klasický OpenGL buffer. Rozdiel spočíva v tom, že ich naviazanie prebieha pomocou odlišných funkcií OpenGL. Rozdiel klasického a „indexovaného“ načítania dát je znázornený na obrázku 3.9.



Obr. 3.9: Názorná ukážka rozdielu pri použití index bufferu (pravá strana). Indexy ukazujú na konkrétny vertex v rámci vertex bufferu. Výhodou je teda potencionálne nižšia pamäťová náročnosť vertex bufferu vďaka odstráneniu duplicitných vertexov.

Vertex Processor

Na vstupe tejto časti sa nachádza pole vertexov. Táto časť je programovateľná pomocou *Vertex shaderu*, ktorého úlohou je transformácia vstupných vrcholov, viď obrázok 3.10. Príklad využitia Vertex shaderu môže byť aplikácia transformačnej matice (zmena veľkosti, rotácia, posun) na pozíciu daného vertexu podľa pohľadu kamery. Výstupom tejto časti je pole spracovaných vertexov.

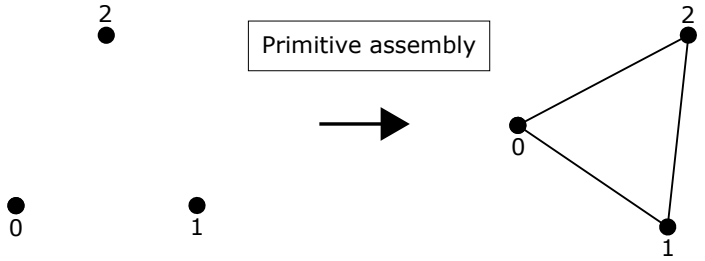


Obr. 3.10: Ukážka použitia Vertex shaderu, ktorý v tomto prípade aplikoval rotačnú maticu na pozíciu jednotlivých vertexov.

Pôvodné atribúty sa ďalšej časti už neodosielajú, no v rámci shaderu je možné deklarovať výstupné premenné, ktoré je možné využiť v shaderi niektorej z ďalších častí. Vertex shader je spustený pre každý vstupný vertex (paralelne).

Primitive assembly

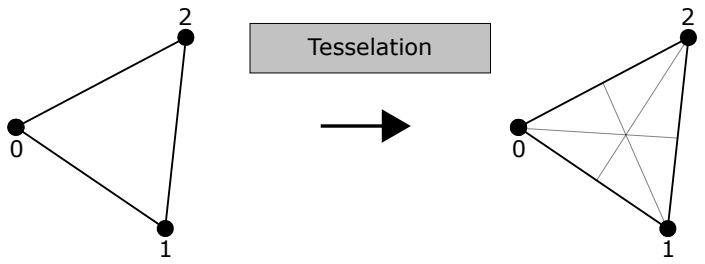
Na vstupe tejto časti sa nachádza opäť pole vertexov. Cieľom tejto časti je na základe konfigurácie kontextu zostaviť primitíva z daných vertexov, viď obrázok 3.11. Typ primitíva (trojuholník, úsečka, štvorec ...) je špecifikovaný v rámci parametrov funkcie, ktorá spúšťa vykresľovanie (draw call). Výstupom tejto časti je množina primitív.



Obr. 3.11: Znázornenie zostavenia trojuholníku pomocou jeho vrcholov.

Tesselation

Teselácia umožňuje rozdeliť primitívny objekt na základe určitých pravidiel, za účelom zvýšenia kvality detailu objektu, viď obrázok 3.12. Vďaka tejto technike je napríklad možné spraviť z pravidelného šesťuholníku definovaného šiestimi vrcholmi hladkú kružnicu, ktorá by za normálnych okolností musela byť definovaná niekoľko násobne vyšším počtom vrcholov. To v konečnom dôsledku značne redukuje nároky na pamäť a potrebný výpočtový výkon vykresľovania. Jedno z použitých pravidiel môže byť napríklad to, ako blízko sa nachádza kamera k danému objektu, na základe čoho zvolíme požadovanú úroveň detailov.



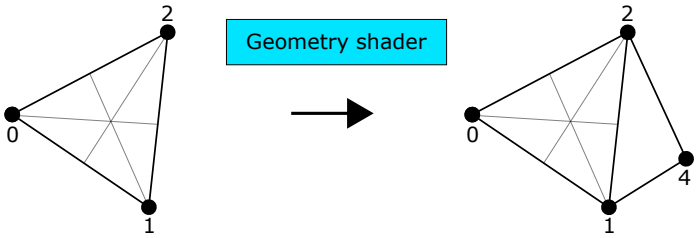
Obr. 3.12: Názorná ukážka teselácie, pomocou ktorej je trojuholník (jedna vstupná primitíva) rozdelená na šesť menších trojuholníkov (šesť výstupných primitív).

Na vstupe tejto časti sa nachádza množina primitív. Táto časť je programovateľná pomocou dvoch typov shaderov. *Tesselation evaluation shader*, pomocou ktorého je možné nastaviť vzor teselácie. Na základe tohto vzoru je primitívny objekt rozdelený na viacero čiastkových primitívnych objektov. *Tesselation control shader* následne tieto primitívne objekty pomocou definovaných pravidiel umiestní do priestoru. Dvojica teselačných shaderov je spustená pre každú vstupnú primitívu (paralelne). Výstupom tejto časti je množina primitív.

Geometry Processor

Na vstupe tejto časti sa nachádza množina primitív. Táto časť je programovateľná pomocou *Geometry shaderu*, pomocou ktorého je možné zahodiť niektoré vstupné primitíva alebo vytvoriť nové na základe toho pôvodného, viď obrázok 3.13. Geometry shader je spustený pre každú vstupnú primitívu (paralelne). Výstupom tejto časti je množina primitív.

Myšlienka Geometry shaderu je podobná teselácii. Geometry shader ale oproti teselácii poskytuje oveľa väčšiu flexibilitu. Jedno z možných využití Geometry shaderu sú napríklad vlasy postavy, ktoré môžu byť definované ako úsečky o počiatočnom a koncovom vertexe. Z týchto bodov môže Geometry shader následne vytvoriť komplexný 3D objekt pripomínajúci reálny vlas. Ďalšie možné využitie sú napríklad časticové systémy, kde jednotlivé častice sú definované bodom v priestore a z týchto bodov Geometry Shader vytvorí komplexnejšie 3D objekty.



Obr. 3.13: Názorná ukážka využitia Geometry shaderu, pomocou ktorého je pridaná jedna nová primitíva.

Primitive Setup

Na vstupe tejto časti sa nachádza množina primitív. Úlohou tejto časti je pripraviť jednotlivé primitíva na rasterizáciu a zahŕňa viacero menších procesov vykonávaných v nasledujúcej postupnosti:

1. **Culling** – Úlohou cullingu je zahodiť odvrátené primitíva. Keďže odvrátené primitíva nie je možné vidieť, nie je dôvod aby boli vykreslené. To či sú jednotlivé primitíva odvrátené alebo nie, vyhodnocuje OpenGL na základe poradia vertexov tvoriacich danú primitívu. Pomocou príslušnej funkcie je možné nastaviť smer vertexov (v smere hodinových ručičiek alebo proti smeru hodinových ručičiek), na základe ktorého sa bude OpenGL rozhodovať.

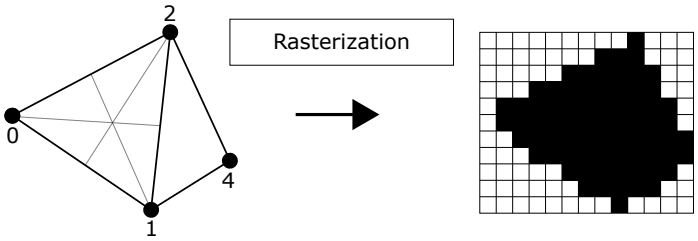
2. **Clipping** – Úlohou clippingu je orezať primitíva, ktoré čiastočne „vytŕčajú“ z NDC (Normalized Device Coordinates) [8, s. 458]. Clipping sa napríklad využíva pri scénach využívajúcich kameru. V takýchto scénach nie je potrebné vykresľovať objekty, ktoré sa nenachádzajú v zábere kamery a teda môžu byť (čiastočne) odstránené.

3. **Perspektívne delenie** – Úlohou perspektívneho delenia je previesť homogénne súradnice do kartézskych súradníc. Pozície všetkých vertexov sú teda vydelené zložkou W, v ktorej je uložená jeho hĺbka.

4. **Viewport transformácia** – Úlohou viewport transformácie je transformovať normalizované súradnice vertexov (NDC) na rozlíšenie obrazovky. Rozlíšenie obrazovky udáva viewport kontextu.

Rasterization

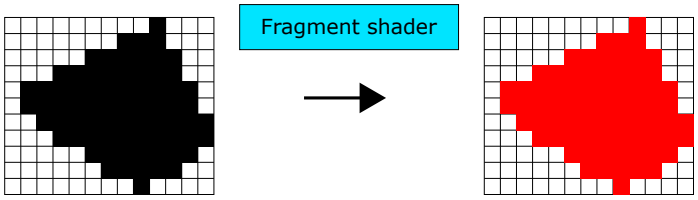
Na vstupe tejto časti sa nachádza množina primitív. Úlohou tejto časti je jednotlivé primitíva rasterizovať. Rasterizácia primitívneho objektu je jeho diskretizácia na fragmenty, teda jednotlivé pixely na obrazovke, viď obrázok 3.14. Výstup tejto časti je množina vytvorených fragmentov.



Obr. 3.14: Znázornenie procesu rasterizácie, ktorý vyplní jednotlivé vstupné primitíva fragmentami. Vytvorené fragmenty nemajú po rasterizácii žiadnu farbu, no na obrázku je použitá čierna pre kontrast. Obecne platí, že čím viac pixelov má obrazovka k dispozícii, tým presnejšia je finálna aproximácia (hladkosť hrán) objektov. V priamej úmere s vyšším rozlíšením ale narastá aj náročnosť na výpočtový výkon.

Fragment Processor

Na vstupe tejto časti sa nachádza množina fragmentov. Táto časť je programovateľná pomocou *Fragment shaderu*, ktorého úlohou je transformácia vstupných fragmentov, viď obrázok 3.15. Fragment shader je spustený pre každý vstupný fragment (paralelne). Fragment shader v rámci výstupu pracuje priamo s práve aktívnym framebufferom.



Obr. 3.15: Ukážka použitia Fragment shaderu, ktorý v tomto prípade priradil jednotlivým fragmentom farbu.

V rámci Vertex shaderu je možné deklarovať výstupné premenné, ktoré je môže použiť vo Fragment shaderi vo forme vstupných premenných (musia mať rovnaký názov). Jednotlivé vertexy tvoriace primitívu môžu mať v konkrétnej výstupnej premennej odlišné hodnoty. Pokiaľ užívateľ označí danú premennú ako *smooth*, sú hodnoty interpolované do jednej výslednej, na základe pozície fragmentu voči jednotlivým vertexom.

Po tejto časti nasledujú v rámci grafickej pipeline ešte dodatočné interné mechanizmy operujúce nad výstupnou množinou fragmentov. Väčšina z týchto mechanizmov nie je pre potreby tejto práce podstatná. Jeden z mechanizmov, ktorý ale stojí za povšimnutie sa nazýva *depth test*, ktorý je možné zapnúť a vypnúť pomocou príslušnej OpenGL funkcie. Tento mechanizmus má za úlohu zabrániť vykresleniu primitív vpredu, zatiaľ čo v rámci priestoru by mali byť skryté za ostatnými.

Depth test si v depth bufferi v rámci aktívneho framebufferu ukladá aktuálnu hĺbku pre každý pixel obrazovky. Pri spracovaní fragmentu sa depth test pozrie na jeho pozíciu

a hĺbku. Hĺbku tohto fragmentu porovná s aktuálne uloženou hĺbkou pre danú v pozíciu v depth bufferi. Pokiaľ je hĺbka tohto fragmentu nižšia ako aktuálne uložená hĺbka, je pre danú pozíciu v color bufferi a depth bufferi aktualizovaná hodnota na farbu a hĺbku tohto nového fragmentu.

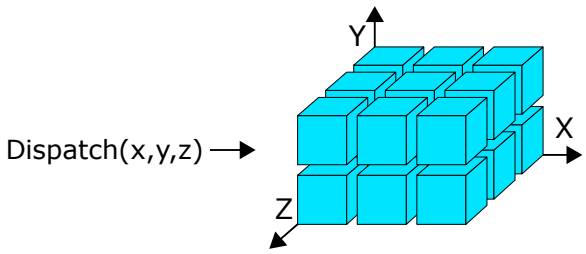
3.2.4 Výpočtová pipeline

V predošlej podsekcii 3.2.3 je popísaná grafická pipeline, ktorá popisuje programovateľný proces vykresľovania. V OpenGL ale existuje ešte aj výpočtová pipeline. Výpočtová pipeline popisuje programovateľný proces všeobecného výpočtu (GPGPU). Na rozdiel od grafickej pipeline, ktorá podporuje rôzne typy shaderov pre programovanie odlišných častí, výpočtová pipeline má iba jednu časť, ktorú je možné programovať jedným shaderom. Tento shader sa nazýva *Compute shader*. Spustenie procesu všeobecného výpočtu sa označuje ako *dispatch*.

Compute Shader

Compute shader nemá žiadne užívateľom definované vstupy ani výstupy. Jediný vstup Compute shaderu je informácia o tom, kde v priestore sa konkrétne spustenie shaderu nachádza. Pokiaľ teda užívateľ chce v rámci Compute shaderu uložiť nejaké dáta tak, aby ich po jeho ukončení bolo možné využiť aj mimo neho, musí využiť SSBO alebo objekty textúr (uniformná premenná typu *image*).

Počet spustení Vertex shaderu v grafickej pipeline závisí od počtu vertexov, počet spustení Fragment shaderu zase od počtu fragmentov. V Compute shaderi je určený *výpočtovým priestorom*. Využíva sa koncept pracovných skupín rozložených v 3D priestore. Veľkosť tohto priestoru (počet pracovných skupín) je špecifikovaný v rámci parametrov funkcie, ktorá spúšťa výpočet (dispatch), viď obrázok 3.16.



Obr. 3.16: Znázornenie výpočtového priestoru. Veľkosť jednotlivých strán je určená v parametroch spustenia. Tento priestor je teda zložený z celkovo šiestich pracovných skupín kde (X, Y, Z) = (3, 2, 2).

Počet pracovných skupín ale ešte neurčuje počet spustení Compute shaderu. Každá pracovná skupina sa ešte delí na lokálny 3D podpriestor, ktorý určuje počet spustení Compute shaderu v rámci pracovnej skupiny. Veľkosť tohto podpriestoru je definovaná v rámci kódu Compute shaderu. Hlavným účelom rozlišovania medzi počtom pracovných skupín a lokálnou veľkosťou je, že rôzne spustenia Compute shaderu v rámci jednej pracovnej skupiny môžu komunikovať prostredníctvom špeciálnych zdieľaných premenných a funkcií. Spustenia v rôznych pracovných skupinách medzi sebou pomocou týchto premenných komunikovať nemôžu, no teoreticky sa dá využiť SSBO (môže spôsobiť deadlock systému). Celkový počet spustení Compute shaderu demonštruje príklad 3.2.

Koncept pracovných skupín môže pripomínať proces spúšťania programu na GPU, ktorý je popísaný v podsekcii 3.1.2. Dalo by sa povedať, že priestor pracovných skupín je abstrakcia thread blocku, ktorý je následne delený na warpy (wavefronty pri AMD). Compute shadery v rámci jedného warpu sú spúšťané sériovo a rôzne warpy paralelne.

Príklad

Konkrétny Compute shader definuje lokálny priestor veľkosti (32, 2, 2). Výpočet je spustený s priestorom pracovných skupín veľkosti (4, 4, 4). Úlohou je vypočítať celkový počet spustení tohto shaderu. Z výpočtu 3.2 vyplýva, že celkový počet spustení tohto shaderu bude 8192.

$$(4.4.4).(32.2.2) = 8192 \tag{3.2}$$

Kapitola 4

Návrh

Táto kapitola popisuje návrh programu popísaného v úvode práce, viď kapitola 1, na základe ktorého bol tento program implementovaný. Princípy popisované v tejto sekcii častokrát obsahujú podobnú terminológiu a teda čitateľovi nemusí byť na prvý pohľad jasný význam konkrétneho termínu, v rámci daného kontextu. Za účelom sprehľadnenia nasledujúcich častí je teda vhodné stanoviť termíny, ktoré budú nadobúdať práve jeden konkrétny význam.

- **Engine** – Program, ktorý je predmetom tejto práce.
- **Užívateľ** – Človek ktorý chce pomocou enginu vytvoriť aplikáciu.
- **Aplikácia** – Užívateľský program, ktorý sa nachádza na vstupe enginu. Obsahuje sekvenciu OpenGL programov a ich shaderov.

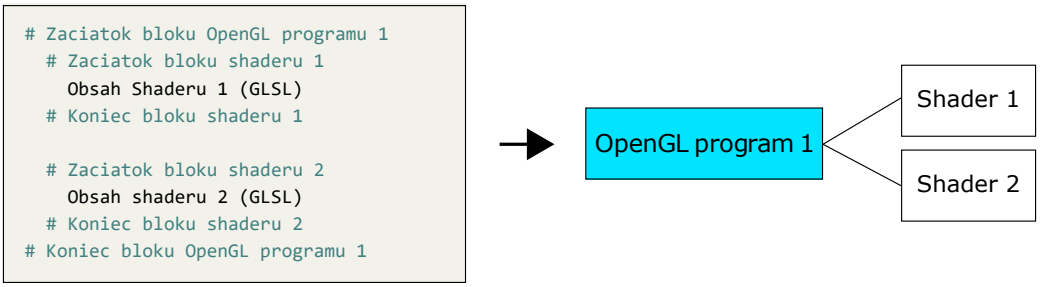
Podrobnejší návrh enginu je rozdelený na viacero častí do samostatných sekcií. Prvá sekcia 4.1 popisuje základný koncept celého enginu. Nasledujúca sekcia 4.2 popisuje formát zdrojového kódu aplikácie. Sekcia 4.3 popisuje spôsob, akým je možné rozšíriť funkcionalitu GLSL shaderov. Sekcia 4.4 popisuje, akým spôsobom je možné spracovať zdrojový kód aplikácie za účelom automatického vytvorenia a konfigurácie potrebných OpenGL objektov. Predposledná sekcia 4.5 popisuje, akým spôsobom je možné spúšťať OpenGL programy špecifikované v rámci aplikácie. Posledná sekcia 4.6 popisuje návrh, vďaka ktorému je možné použiť myš a klávesnicu priamo na GPU zo shaderov.

4.1 Základný koncept

Úlohou enginu je načítať aplikáciu a skompilovať ju. V prípade úspešného výsledku kompilácie je aplikácia spustená, teda je užívateľovi otvorené okno obsahujúce výstupný obraz. Engine by mal byť spustiteľný súbor, ktorý je možné spustiť pomocou príkazovej riadky a jeho použitie by malo byť jednoduché. Jednou z podmienok jednoduchosti použitia je minimálny počet parametrov príkazovej riadky pri jeho spúšťaní. Z tohto dôvodu by mal mať engine iba jeden, povinný parameter, pomocou ktorého môže užívateľ špecifikovať cestu k súboru, v ktorom sa nachádza zdrojový kód aplikácie. Samotný engine musí pochopiteľne bežať na CPU, no aplikácia spúšťaná týmto enginom bude bežať na GPU.

4.2 Formát zdrojového kódu aplikácie

Aby bolo možné navrhnúť formát zdrojového kódu je potrebné si uvedomiť, ako fungujú rôzne grafické programy využívajúce OpenGL. Pre vykreslenie objektu je potrebné vytvoriť OpenGL program, ktorý tento objekt pomocou rôznych shaderov spracuje a vykreslí. Rôzne objekty môžu vyžadovať odlišné spracovanie, teda je potrebné vytvoriť rôzne shadery a programy, ktoré budú prepínané podľa toho, aký objekt treba vykresliť. Keďže sú shadery od seba nezávislé, typicky sa nachádzajú v osobitnom súbore. Neprichádza teda do úvahy, aby užívateľ nahrával do enginu každý shader osobitne. Užívateľ musí mať možnosť špecifikovať všetky shadery v rámci jedného súboru. Aby engine mohol pri spustení tento súbor spätne rozdeliť do jednotlivých shaderov, je potrebné, aby sa každý program a jeho shadery nachádzali v nejakom bloku, ktorý by ho logicky oddeľoval od ostatných, viď obrázok 4.1.



Obr. 4.1: Znázornenie štruktúry zdrojového kódu aplikácie. Pomocou deklarácii začiatku a konca bloku môže engine spätne skonštruovať jednotlivé OpenGL programy a priradiť im ich shadery. V rámci deklarácie začiatku bloku shaderu musí byť špecifikovaný aj jeho typ.

Takto navrhnutý formát vstupu umožňuje definovať sekvenciu jedného až viacerých OpenGL programov. Zároveň umožňuje pre každý OpenGL program definovať shadery tak, že ich kód nekoliduje s kódom iného shaderu. To všetko v rámci jedného súboru, ktorého cesta sa nachádza na vstupe enginu ako jediný parameter.

Konfiguračné parametre

Ako už bolo spomenuté, engine nemá žiadne argumenty príkazovej riadky mimo špecifikácie cesty k zdrojovému kódu aplikácie. Z tohto dôvodu užívateľ nebude schopný v rámci spustenia enginu špecifikovať žiadne konfiguračné parametre, ktoré by prispôsobili jeho chovanie podľa potreby aplikácie. Užívateľ môže chcieť napríklad zmeniť šírku a výšku vykresľovacieho okna, názov okna, prípadne zapnúť rôzne voliteľné moduly pre potreby jeho aplikácie. Užívateľ teda musí mať možnosť špecifikovať jednotlivé parametre v rámci zdrojového kódu aplikácie. V aplikácii sa môžu vyskytnúť dva typy konfiguračných parametrov:

- **Parameter aplikácie** – parameter ktorý konfiguruje celú aplikáciu,
- **Parameter OpenGL programu** – parameter ktorý konfiguruje jeden konkrétny OpenGL program v rámci aplikácie.

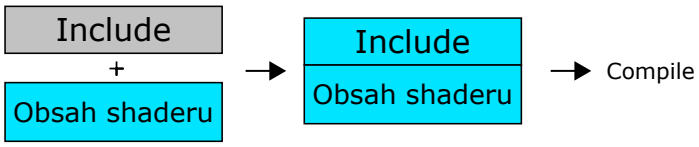
Intuitívny spôsob deklarácie parametrov by teda mal vyzerať tak, že pokiaľ ide o parameter OpenGL programu, tento parameter sa bude nachádzať v bloku OpenGL programu. Pokiaľ sa nejaký parameter nachádza mimo akéhokoľvek bloku, jedná sa o parameter aplikácie, viď znázornenie 4.1.

Výpis 4.1: Znázornenie spôsobu deklarácie konfiguračných parametrov aplikácie a jednotlivých programov.

```
% Parameter aplikacie - Typ - Hodnota

# Zaciatok bloku OpenGL programu 1
    % Parameter programu - Typ - Hodnota

    # Zaciatok bloku shaderu 1
        Obsah shaderu 1 (GLSL)
    # Koniec bloku shaderu 1

    # Zaciatok bloku shaderu 2
        Obsah shaderu 2 (GLSL)
    # Koniec bloku shaderu 2
# Koniec bloku OpenGL programu 1
```

4.3 Rozšírené možnosti shaderu

Jedným z cieľov enginu je poskytnúť užívateľovi rozšírenú funkcionalitu, ktorá nie je na GPU alebo v rámci OpenGL dostupná. Tieto nové funkcie by mal byť užívateľ schopný použiť z jednotlivých shaderov aplikácie. Môže sa jednať o nové GLSL funkcie, konštanty alebo rôzne vstavané deklarácie. V klasických programovacích jazykoch pre CPU sa toto rieši tak, že nástroj alebo knižnica disponuje súbormi obsahujúcimi zdrojový kód, ktoré si môže programátor vložiť do svojho zdrojového kódu pomocou vkladacieho príkazu daného jazyka (include). Jazyk GLSL ale nepodporuje takéto vkladanie súborov, nakoľko GPU nemá súborový systém. Vyriešiť sa to dá tak, že engine pred kompiláciou jednotlivých shaderov manuálne vloží zdrojový kód obsahujúci nové funkcie, konštanty a rôzne vstavané deklarácie na začiatok zdrojového kódu daného shaderu, viď obrázok 4.2.



Obr. 4.2: Znázornenie vloženia kódu na začiatok zdrojového kódu shaderu. Užívateľ teda môžu efektívne využívať funkcionalitu vloženého kódu v rámci pôvodného obsahu shaderu, ako keby sa nachádzala natívne priamo v GLSL. Shader bude skompilovaný až po spojení.

4.4 Vytvorenie potrebných objektov

V prechádzajúcej sekcii 4.2 je popísané, akým spôsobom môže užívateľ špecifikovať shadery v rámci aplikácie. Jednotlivé shadery treba skompilovať pomocou OpenGL a naviazať ich na odpovedajúce OpenGL programy, ktoré budú spúšťané. Ako už bolo spomenuté v úvode tejto práce, viď kapitola 1, shader predpokladá že OpenGL objekty ktoré počas behu využíva, sú vytvorené a správne nakonfigurované. Úlohou enginu je teda tieto OpenGL objekty na základe analýzy zdrojového kódu automaticky vytvoriť a nakonfigurovať, aby shadery využívajúce tieto objekty mohli byť úspešne spustené. Cieľom tejto sekcie je ukázať, ako

je možné využiť zdrojový kód aplikácie, za účelom automatického vytvárania potrebných OpenGL objektov.

OpenGL program je možné kategorizovať dvoma spôsobmi podľa toho, aké typy shaderov obsahuje:

- **Vykresľovací program** – Obsahuje shadery využívané v grafickej pipeline, viď podsekcia 3.2.3.
- **Výpočtový program** – Obsahuje shadery využívané vo výpočtovej pipeline, viď podsekcia 3.2.4.

Kategorizovať jednotlivé OpenGL programy je teda v princípe jednoduché. V prípade že OpenGL program obsahuje Compute shader, jedná sa o výpočtový program. V opačnom prípade sa jedná o vykresľovací program. V prípade že by OpenGL program obsahoval Compute shader a aj shadery využívané v grafickej pipeline, jeho vytvorenie zlyhá. Dôvod tejto kategorizácie je fakt, že vykresľovací program vyžaduje vytvorenie a konfiguráciu niektorých špeciálnych OpenGL objektov, ktoré výpočtový program nevyžaduje. Jedná sa o VAO a framebuffer. Ostatné základné OpenGL objekty ako napríklad textúry a SSBO môžu byť použité nezávisle od kategórie programu a typu shaderu.

Čiastočná limitácia enginu spočíva v tom, že nie je možné použiť uniformné premenné za účelom zasielania vlastných dát. Pomocou uniformných premenných môže aplikácia bežiaca na CPU poslať shaderom dáta, viď podsekcia 3.2.1. Keďže užívateľ nebude do CPU časti zasahovať, nebude môcť uniformné premenné využiť. Namiesto uniformných premenných ale môže použiť SSBO, takže je možné docieliť rovnakej funkcionality, akurát odlišným spôsobom. Výnimkou sú uniformné premenné typu *image* a *sampler*, pomocou ktorých je možné pracovať s textúrami v rámci shaderu, viď podsekcia 3.2.1. Použitie uniformných premenných tohto typu musí byť enginom podporované.

Nasledujúce sekcie teda popisujú spôsob detekcie, vytvorenia a konfigurácie vyššie zmienených OpenGL objektov.

Detekcia a vytvorenie SSBO

Na ukážke kódu 4.2 je znázornený spôsob deklarácie SSBO bloku v GLSL. Aby pomocou tohto bloku bolo možné pracovať s danými dátami, je potrebné vytvoriť buffer a následne tento buffer pred spustením programu naviazať na binding point odpovedajúceho SSBO bloku. Pokiaľ teda užívateľ pracuje s premennými SSBO bloku, pracuje s bufferom, ktorý je naviazaný na jeho binding point, teda dva rôzne SSBO bloky musia mať rozdielne binding pointy.

Výpis 4.2: Ukážka spôsobu deklarácie SSBO bloku v GLSL. Deklarácia sa skladá z layout parametrov, názvu bufferu a premenných tvoriacich tento blok. Prvý layout parameter udáva spôsob rozloženia dát v pamäti. Druhý parameter udáva binding point tohto bloku.

```
layout(std430, binding = 0) buffer mojBuffer {
    type premenna1;
    type premenna2;
    type premenna3;
};
```

Na základe deklarácie SSBO bloku musí byť vytvorený buffer. Názov SSBO bloku môže byť „zneužitý“ ako unikátny identifikátor bufferu. Pokiaľ sa teda bude v dvoch rôznych shaderoch nachádzať SSBO blok s rovnakým názvom, obidva ukazujú na ten istý buffer. Engine si teda musí uchovávať informáciu o doposiaľ vytvorených bufferoch a ich názvoch. Ďalej si engine musí pre každý OpenGL program uchovávať informáciu o jednotlivých SSBO blokoch v jeho shaderoch a ich binding pointoch, aby ich pred spustením mohol na odpovedajúce biding pointy naviazať.

Aby bolo možné vytvorený buffer použiť, je potrebné inicializovať jeho veľkosť v pamäti. Tú je možné vypočítať automaticky sčítaním pamäťovej náročnosti typov jednotlivých premenných bloku. Takto automatické sčítanie ale nemusí byť vždy presné[1] a teda by užívateľ navyše mal mať možnosť túto veľkosť špecifikovať sám. Požadovaná veľkosť v bytoch môže byť teda obsiahnutá v názve bloku, napr. `mojBuffer_30`. V prípade že užívateľ túto veľkosť nešpecifikuje, bude použitý automatický výpočet.

Detekcia a vytvorenie objektov textúr

Na ukážke kódu 4.3 je znázornený spôsob deklarácie uniformnej premennej typu *image* alebo *sampler* v GLSL[2]. Pomocou uniformnej premennej tohto typu môže užívateľ pracovať s vytvorenými textúrami.

Výpis 4.3: Ukážka spôsobu deklarácie uniformnej premennej. Deklarácia sa skladá z typu uniformnej premennej a jej názvu.

```
uniform sampler2D textura1;
uniform image2D textura2;
```

Na základe tejto deklarácie musí byť vytvorený OpenGL objekt textúry. Názov uniformnej premennej môže byť opäť „zneužitý“ ako unikátny identifikátor konkrétneho OpenGL objektu textúry. Pokiaľ sa bude v dvoch rôznych shaderoch nachádzať uniformná premenná s rovnakým názvom, obidve ukazujú na ten istý objekt. Engine si teda podobne ako pri SSBO musí uchovávať informáciu o doposiaľ vytvorených objektoch textúr a ich názvoch. Taktiež si musí pre každý OpenGL program uchovávať informáciu o jednotlivých uniformných premenných v jeho shaderoch, aby na ne mohol pred spustením naviazať odpovedajúce OpenGL objekty textúr.

OpenGL objekt textúry môže mať viacero nastavení, ktoré by mal mať užívateľ možnosť prispôsobiť. Základné nastavenie je jej výška a šírka v pixeloch. Z tohto dôvodu musí názov uniformnej premennej obsahovať informáciu o týchto rozmeroch, aby ju engine vedel nakonfigurovať, napr. `textura1_300x300`.

Vytvorenie a konfigurácia VAO

Pokiaľ je OpenGL program vykresľovací, je potrebné vytvoriť a nakonfigurovať VAO, ktoré popisuje formát a zdroj vstupných dát grafickej pipeline, viď časť Vertex Assembly v podsekcii 3.2.3. Na ukážke kódu 4.4 je znázornený spôsob deklarácie vstupných dát v rámci Vertex shaderu.

[1] SSBO blok umožňuje deklarovať pole o neurčitej veľkosti, ktoré využije celý zvyšný dostupný priestor naviazaného bufferu.

[2] Súčasťou tohto typu je informácia o počte dimenzií, teda napríklad image1D, image2D, image3D a podobne aj pre sampler.

Výpis 4.4: Ukážka spôsobu deklarácie vstupných premenných shaderu. Deklarácia sa skladá z typu premennej a jej názvu. V rámci deklarácie je možné špecifikovať aj layout parameter lokácie. Parameter lokácie v prípade VAO určuje, ktorý jeho atribút bude naviazaný na danú premennú.

```
layout (location = 0) in type premenna1;
layout (location = 1) in type premenna2;
```

Proces vytvárania VAO funguje v bežných programoch tak, že sa na základe štruktúry dát nejakého objektu vytvorí VAO a až následne sa vo Vertex shaderi (ktorý spracováva daný objekt) na základe tohto VAO deklarujú vstupné premenné. V prípade enginu je ale potrebné postupovať opačne, keďže musí byť VAO vytvorené na základe deklarácií v zdrojovom kóde.

Pre každý vykresľovací OpenGL program musí byť teda vytvorené osobitné VAO. V ňom musí byť na základe každej vstupnej premennej vytvorený atribút s odpovedajúcim typom a lokáciou. Na záver musí byť na toto VAO naviazaný buffer v ktorom sa nachádzajú dáta. Užívateľ teda pomocou konfiguračného parametru OpenGL programu, viď sekcia 4.2, špecifikuje názov SSBO bloku, ktorý predstavuje vertex buffer. Rovnakým spôsobom môže v prípade potreby nastaviť aj index buffer. Engine si musí uchovávať informáciu o tom, ktoré VAO patrí ktorému OpenGL programu.

Vytvorenie a konfigurácia framebufferu

Pokiaľ chce užívateľ vykresľovať na obrazovku (do okna), je použitý predvolený framebuffer, ktorý je vytvorený a nakonfigurovaný už v rámci vytvárania OpenGL kontextu, viď podsekcia 3.2.1. Pokiaľ chce ale užívateľ vykresľovať do OpenGL textúry, je potrebné vytvoriť a nakonfigurovať nový framebuffer a naviazať naň danú textúru. Na ukážke kódu 4.5 je možné vidieť spôsob deklarácie výstupu Fragment shaderu (posledná časť grafickej pipeline pracujúca s framebufferom).

Výpis 4.5: Ukážka spôsobu deklarácie výstupných predmetov shaderu. Deklarácia sa skladá z typu premennej a jej názvu.

```
out vec4 farba;
```

Engine musí podporovať konfiguračný parameter OpenGL programu, pomocou ktorého mu dá užívateľ vedieť, že nemá využiť predvolený framebuffer. V takomto prípade je nutné vytvoriť nový framebuffer. Výstupné premenné Fragment shaderu sa v tomto prípade budú považovať za objekty textúr. V jednej z predošlých častí je popísaný spôsob vytvárania a konfigurácie textúr, ktorý bude využitý aj v tomto prípade. Užívateľ teda môže napríklad pomocou jedného OpenGL programu vykresliť obraz do textúry a v nasledujúcom OpenGL programe túto textúru načítať pomocou uniformnej premennej. Engine si musí uchovávať informáciu o tom, ktorý framebuffer patrí ktorému OpenGL programu.

4.5 Spustenie jednotlivých programov

V predchádzajúcej sekcii 4.4 je popísaný spôsob vytvárania OpenGL objektov za účelom úspešného spustenia jednotlivých OpenGL programov tvoriacich aplikáciu. Táto sekcia popisuje spôsob, akým sú tieto OpenGL programy v engine spúšťané.

Spustiť OpenGL program znamená aktivovať daný OpenGL program a spustiť vykresľovanie (draw call) alebo výpočet (dispatch) pomocou príslušnej OpenGL funkcie. Tieto funkcie typicky obsahujú rôzne parametre. V prípade vykresľovania sa typicky jedná o špecifikáciu typu primitív a počtu vertexov. V prípade výpočtu ide o špecifikáciu počtu pracovných skupín. Problém je, že tieto príkazy spúšťa engine a nie užívateľ.

Pre vyriešenie tohto problému v prípade vykresľovania, musí engine vytvoriť „vstavaný“ buffer, v ktorom budú uložené požiadavky na vykresľovanie. Pomocou mechanizmu popísaného v sekcii 4.3 bude shaderom poskytnutá deklarácia SSBO bloku, na ktorý bude tento buffer naviazaný. Jediná limitácia užívateľa tohto návrhu spočíva v tom, že prvý OpenGL program aplikácie musí byť výpočtový program, ktorý pomocou Compute shaderu inicializuje toto SSBO do požadovaného stavu pre úspešné spustenie nasledujúcich vykresľovacích programov.

V prípade výpočtu je tento problém možné vyriešiť identicky. Engine vytvorí vstavaný buffer, ktorý bude obsahovať počet pracovných skupín. Shaderom bude poskytnutá deklarácia SSBO bloku, na ktorý bude tento buffer naviazaný. Vzniká identická limitácia ako pri riešení problému v prípade vykresľovania, ktorú je nutné vyriešiť odlišne. Predvolený počet pracovných skupín engine nastaví na jednu, aby bol po spustení prvého výpočtového programu spustený minimálne aspoň jeden Compute shader.

4.6 Podpora myši a klávesnice

Zatiaľ čo z programov spúšťaných na CPU je možné pracovať so všetkými komponentami pripojenými do PC pomocou rôznych rozhraní, funkcionalita GPU a GLSL je v tomto ohľade podstatne limitovaná, keďže v programoch bežiacich na GPU môžeme pracovať „iba“ so samotnou GPU. Aby mohol užívateľ tvoriť pomocou enginu komplexnejšie aplikácie, je potrebné navrhnúť spôsob, akým by užívateľ mohol v rámci aplikácie používať základné periférie ako napríklad myš a klávesnica.

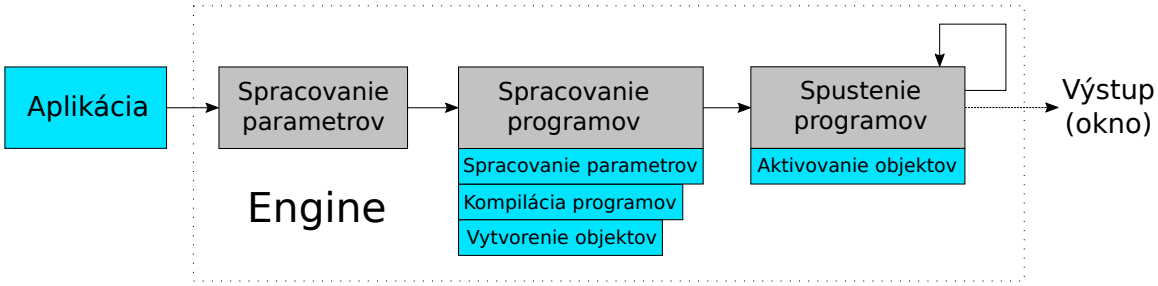
Za týmto účelom musí engine detekovať stlačené klávesy, tlačidlá myši a pozíciu myši a ukladať ich aktuálny stav. Aby k tomuto stavu mohol pristupovať užívateľ z jednotlivých shaderov, je potrebné vytvoriť vstavaný buffer, ktorý bude uchovávať aktuálny stav týchto periférií. Pomocou mechanizmu popísaného v sekcii 4.3 bude shaderom poskytnutá deklarácia SSBO bloku, na ktorý bude tento buffer naviazaný. Poskytnuté budú aj rôzne funkcie, ktoré abstrahujú prácu s týmto bufferom. Môže to byť napríklad funkcia ktorá vráti bool v závislosti od toho, či bola daná klávesa stlačená. Na rovnakom princípe by mohli byť podporované aj iné periférie.

Tento buffer môže byť rovno využitý aj k ukladaniu iných informácii enginu, ktoré môže chcieť užívateľ použiť. Môže sa jednať napríklad o aktuálne rozmery okna, aktuálny čas, prípadne rozdiel času od posledného snímku (delta time). Tento buffer bude teda nazývaný ako *engine buffer*.

Kapitola 5

Implementácia

Táto kapitola popisuje spôsob implementácie enginu, ktorého návrh je možné nájsť v kapitole 4. Na obrázku 5.1 je možné vidieť znázornenie, ako implementovaný engine funguje ako celok. Implementácia enginu je pomerne komplexná a teda náročná na vysvetlenie tak, aby bolo jasné ako čo funguje a v akom poradí sa to vykonáva. Z tohto dôvodu je najprv vysvetlený spôsob implementácie niektorých dôležitých a zložitejších procesov a až následne je popísaná celková štruktúra programu. Sekcia 5.1 popisuje implementáciu spracovania konfiguračných parametrov. Sekcia 5.2 popisuje spracovanie zdrojového kódu aplikácie za účelom vytvorenia OpenGL programov a kompilácie jeho shaderov. Sekcia 5.3 popisuje implementáciu spúšťania programov. Predposledná sekcia 5.4 popisuje rozšírenie jazyka GLSL o nové funkcie a posledná sekcia 5.5 následne popisuje celkovú štruktúru programu.



Obr. 5.1: Znázornenie postupnosti spúšťania jednotlivých celkov enginu. Na vstupe enginu sa nachádza aplikácia. Z jej zdrojového kódu sú spracované konfiguračné parametre aplikácie. Nasleduje spracovanie jednotlivých OpenGL programov, ktoré sú následne dookola spúšťané až do ukončenia enginu.

Engine bol implementovaný v programovacom jazyku `C++20`. Pre prácu s OpenGL bola využitá knižnica `GLEW`[1] (OpenGL Extension Wrangler). Pre vytvorenie OpenGL kontextu, okna operačného systému, pre detekciu stlačených kláves a pohyb myši bola využitá knižnica `GLFW`[2] (Graphics Library Framework).

[1] http://glew.sourceforge.net
[2] https://glfw.org

5.1 Spracovanie konfiguračných parametrov

Pre spôsob deklarácie konfiguračných parametrov, viď sekcia 4.2, bola zvolená direktíva preprocesoru `#pragma`. Originálny účel tejto direktívy je odosielanie dodatočných informácii prekladaču jazyka GLSL. V prípade že sa za direktívou nachádza hodnota, ktorú prekladač nepodporuje, je tento riadok ignorovaný a teda nijako neovplyvní priebeh kompilácie. Z tohto dôvodu je to ideálna konštrukcia, nakoľko nijako neovplyvní sémantický význam zdrojového kódu shaderov. Užívateľ teda môže v rámci zdrojového kódu aplikácie deklarovať konfiguračný parameter aplikácie v nasledujúcom formáte:

```
#pragma PARAM typ_parametru hodnota;
```

Extrahovanie konfiguračných parametrov deklarovaných týmto spôsobom bolo implementované pomocou regulárneho výrazu. Pre prácu s regulárnymi výrazmi bola využitá štandardná knižnica `<regex>` jazyka `C++`. V rámci hodnoty parametru umožňuje regulárny výraz zadať sekvenciu znakov bez medzery, prípadne sekvenciu znakov s medzerou ohraničenú úvodzovkami (reťazec).

V prípade konfiguračných parametrov OpenGL programu, bude tento formát trochu odlišný. Keďže je pre extrahovanie konfiguračných parametrov využitý regulárny výraz, je obtiažne zistiť jeho pozíciu v zdrojovom kóde voči blokom programov a teda určiť v ktorom programe sa nachádza. Kľúčové slovo PARAM teda musí obsahovať identifikátor programu, ku ktorému má byť tento parameter pridelený. Konkrétny spôsob identifikácie je popísaný v rámci nasledujúcej sekcie 5.2, popisujúcej spracovanie OpenGL programov.

5.2 Spracovanie programov

Pre spôsob deklarácie začiatku a konca blokov, viď sekcia 4.2, bola zvolená direktíva preprocesoru `#ifdef hodnota` a `#endif`. Účel týchto direktív je podmienečná kompilácia kódu, ktorý sa nachádza medzi nimi a je vyhodnotená ešte pred samotnou kompiláciou [4]. Pokiaľ teda nie je definovaná daná hodnota pomocou direktívy preprocesoru `#define hodnota`, daný kód nebude skompilovaný.

Užívateľ teda môže v rámci zdrojového kódu aplikácie deklarovať začiatok bloku konkrétneho OpenGL programu a shaderu následujúcim spôsobom:

```
#ifdef PROGRAM_(ID)
```

Výpis 5.1: Ukážka formátu deklarácie začiatku bloku OpenGL programu. `ID` predstavuje unikátnu identifikáciu programu v rámci aplikácie, reprezentovanú nezáporným číslom.

```
#ifdef PROGRAM_(ID)_(TYPE)
```

Výpis 5.2: Ukážka formátu deklarácie začiatku bloku shaderu. `ID` priraďuje shader konkrétnemu OpenGL programu. TYPE udáva informáciu o tom, akého je shader typu a môže nadobúdať hodnoty: `COMPUTE_SHADER`, `VERTEX_SHADER`, `FRAGMENT_SHADER`, `GEOMETRY_SHADER`, `TESS_CONTROL_SHADER` alebo `TESS_EVALUATION_SHADER`.

Pre zistenie toho, aké OpenGL programy sa v zdrojovom kóde aplikácie nachádzajú, bol využitý regulárny výraz hľadajúci všetky výskyty textu deklarácie začiatku bloku OpenGL programu, za účelom získania ich `ID`. Pre nájdenie konfiguračných parametrov jednotlivých

OpenGL programov bol použitý regulárny výraz hľadajúci výskyty nasledujúceho reťazcu:

```
#pragma PROGRAM_(ID)_PARAM typ_parametru hodnota
```

Pre každý nájdený OpenGL program teda engine vytvorí inštanciu triedy `Program`, do ktorej uloží jeho `ID`, konfiguračné parametre a odkaz na inštanciu triedy `Engine`. Inštancie triedy `Program` si engine ukladá v atribúte `programs`. Nad každou inštanciou je spustená metóda `compile`, ktorej úlohou je kompilácia shaderov daného OpenGL programu, viď podsekcia 5.2.1 a vytvorenie potrebných OpenGL objektov, viď podsekcia 5.2.2.

5.2.1 Kompilácia OpenGL programu

Na vstupe metódy `compile` triedy `Program` sa nachádza celý zdrojový kód aplikácie. Pre nájdenie shaderov daného OpenGL programu bol využitý regulárny výraz hľadajúci pomocou `ID` všetky výskyty textu deklarácie začiatku bloku shaderu. Nájdené shadery je možné skompilovať zaujímavým spôsobom. Vďaka využitiu zmienených direktív preprocesoru `#ifdef` a `#endif` je možné zakaždým kompilovať celý zdrojový kód aplikácie. Stačí že engine na začiatok zdrojového kódu vloží direktívu preprocesora `#define` tak, aby bol predmetom kompilácie iba konkrétny shader. Pred kompiláciou zdrojového kódu sú zároveň na jeho začiatok vložené vstavané deklarácie enginu, rozširujúce funkcionalitu GLSL, viď sekcia 4.3.

Výpis 5.3: Názorná ukážka použitia direktívy preprocesora define tak, že predmetom kompilácie bude iba obsah konkrétneho Vertex shaderu.

```
#define PROGRAM_0
#define PROGRAM_0_VERTEX_SHADER

#ifdef PROGRAM_0
    #ifdef PROGRAM_0_VERTEX_SHADER
        Obsah Vertex shaderu (GLSL)
    #endif

    #ifdef PROGRAM_0_FRAGMENT_SHADER
        Obsah fragment shaderu (GLSL)
    #endif
#endif
```

OpenGL objekt shaderu je vytváraný pomocou funkcie `glCreateShader`. Zdrojový kód do tohto objektu je načítaný pomocou funkcie `glShaderSource`. Shader je na záver preložený pomocou funkcie `glCompileShader`. Pomocou funkcie `glGetShaderiv` je verifikovaný úspešný stav preloženia. Engine teda nemusí vykonávať vlastnú syntaktickú a sémantickú analýzu nad jednotlivými shadermi, nakoľko sa o to stará OpenGL.

Po úspešnom skompilovaní všetkých shaderov je vytvorený objekt OpenGL programu pomocou funkcie `glCreateProgram`. Jednotlivé objekty shaderov sú naň pripojené pomocou funkcie `glAttachShader`. Objekt OpenGL programu je následne naviazaný pomocou funkcie `glLinkProgram` a validovaný funkciou `glValidateProgram`. Úspešný výsledok naviazania a validácie je overený pomocou funkcie `glGetShaderiv`. V atribúte `program` triedy `Program` je uchovávané `ID` vytvoreného objektu OpenGL programu.

5.2.2 Vytvorenie OpenGL objektov

Návrh spôsobu spracovania zdrojového kódu za účelom vytvorenia OpenGL objektov potrebných pre úspešné spustenie programu je popísaný v sekcii 4.4. Jeden možný spôsob implementácie spočíval v použití regulárnych výrazov pre extrahovanie potrebných informácií o jednotlivých konštrukciách. Vznikol by ale rovnaký problém ako pri spracovaní konfiguračných parametrov spočívajúci v zistení pozície v zdrojovom kóde voči blokom shaderov a teda určiť v ktorom shaderi sa nachádza. OpenGL našťastie poskytuje rôzne funkcie, ktoré vedia extrahovať všetky potrebné informácie zo skompilovaných programov a teda nebolo nutné implementovať manuálne spracovanie zdrojového kódu pre získanie týchto informácií. Medzi tieto funkcie patrí:

- `glGetProgramInterfaceiv` – Funkcia pre dotazovanie sa na vlastnosť rozhrania programu. Napríklad celkový počet aktívnych uniformných premenných programu alebo vstupných a výstupných premenných vykresľovacieho programu.
- `glGetProgramResourceName` – Funkcia, ktorá vráti názov zdroja nachádzajúceho sa na nejakom indexe[3]. Typicky sa používa v kombinácii s predchádzajúcou funkciou, ktorá vráti počet zdrojov tohto typu.
- `glGetProgramResourceIndex` – Funkcia, ktorá vráti index zdroja na základe jeho názvu.
- `glGetProgramResourceiv` – Funkcia, ktorá vracia rôzne vlastnosti zdroja nachádzajúceho sa na nejakom indexe. Vlastnosti ktoré je možné získať záležia od typu daného zdroja. Pomocou tejto funkcie je napr. možné zistiť binding location a pamäťovú náročnosť SSBO.
- `glGetActiveUniform` – Funkcia, ktorá vráti informácie o uniformnej premennej na danom indexe, napríklad jej názov a typ.
- `glGetUniformLocation` – Funkcia, ktorá vráti lokáciu uniformnej premennej[4] na základe jej názvu.

Vytvorenie SSBO

Vytváranie SSBO je založené na návrhu ich detekcie a konfigurácie, popísaného v rámci sekcie 4.4. Pre vytvorenie odpovedajúceho bufferu disponuje trieda `Engine` pomocnou metódou `createBuffer(name, size)`. Pomocou regulárneho výrazu je spracovaný názov špecifikovaný v rámci parametru `name`, ktorý môže obsahovať požadovanú veľkosť v bytoch. V prípade že nebola veľkosť v rámci názvu špecifikovaná, je využitá veľkosť z parametru `size`. Pamäťový priestor bufferu je teda inicializovaný na požadovanú veľkosť pomocou funkcie `glNamedBufferData`. `ID` vytvoreného bufferu je uložené spolu s jeho názvom do atribútu `buffers` triedy `Engine`. V prípade že buffer s daným názvom už existuje, metóda vracia `ID` už existujúceho.

Pre vytvorenie bufferov je najprv pomocou funkcie `glGetProgramInterfaceiv` potrebné zistiť počet aktívnych SSBO blokov OpenGL programu. Pre každý SSBO blok je pomocou funkcie `glGetProgramResourceName` zistený jeho názov a pomocou funkcie

[3] Pomocou indexu sa indexujú zdroje programu daného typu za účelom získania informácii o jeho vlastnostiach.

[4] V prípade uniformnej premennej sa jej pomocou lokácie priraďuje hodnota

`glGetProgramResourceiv` jeho pamäťová náročnosť a binding point. V atribúte `buffers` triedy `Program` sú uchovávané väzby medzi binding pointom SSBO bloku a návratovou hodnotou funkcie `createBuffer`, ktorá vráti `ID` bufferu. Engine tento atribút neskôr využije, aby pred spustením daného OpenGL programu vedel, aké buffery naviazať a na aký binding point.

Vytvorenie Textúr

Vytváranie textúr je založené na návrhu ich detekcie a konfigurácie, popísaného v rámci sekcie 4.4. Pre vytvorenie objektov textúr disponuje trieda `Engine` pomocnou metódou `createTexture(name)`. Objekt textúry je vytvorený pomocou funkcie `glCreateTextures`. Pomocou regulárneho výrazu je spracovaný názov špecifikovaný v rámci parametru `name`, ktorý obsahuje požadované rozmery textúry. Pamäťový priestor textúry je inicializovaný na požadované rozmery pomocou funkcie `glTextureStorage2D`. `ID` vytvorenej textúry je uložený spolu s jej názvom do atribútu `textures` triedy `Engine`. V prípade že textúra s daným názvom už existuje, metóda vracia `ID` už existujúcej.

Pre vytvorenie textúr je najprv pomocou funkcie `glGetProgramInterfaceiv` potrebné zistiť počet aktívnych uniformných premenných OpenGL programu. Pre každú uniformnú premennú je pomocou funkcie `glGetActiveUniform` zistený jeho názov a dátový typ. Pomocou funkcie `glGetUniformLocation` je zistená lokácia danej uniformnej premennej. Podporované typy uniformných premenných sú iba image a sampler. V atribúte `textures` triedy `Program` sú uchovávané väzby medzi lokáciou uniformnej premennej, jej typom a návratovou hodnotou funkcie `createTexture`, ktorá vráti `ID` textúry.

Vytvorenie VAO

Vytvorenie VAO je založené na návrhu jeho vytvorenia a konfigurácie, popísaného v rámci sekcie 4.4. V prípade že daný OpenGL program nie je vykresľovací, vytvorenie VAO nieje potrebné. V opačnom prípade je najprv pomocou funkcie `glCreateVertexArrays` vytvorené VAO. Pomocou funkcie `glGetProgramInterfaceiv` je zistený počet aktívnych vstupných premenných OpenGL programu (Vertex shaderu). Pre každú vstupnú premennú definovanú užívateľom je zistený jej typ a lokácia pomocou funkcie `glGetProgramResourceiv`. Na základe jej lokácie je vo vytvorenom VAO zapnutý atribút na danej lokácii pomocou funkcie `glEnableVertexArrayAttrib`. Na základe jej typu je tento atribút naformátovaný pomocou funkcie `glVertexArrayAttribFormat`. Táto funkcia očakáva špecifikáciu typu pomocou skalárneho typu a jeho počtu. Z tohto dôvodu disponuje trieda `Utils` pomocnými funkciami, ktoré prevedú rôzne zložené typy, ako napríklad vektor, do požadovaného tvaru.

Pomocou funkcie `glVertexArrayVertexBuffer` je vytvorenému VAO priradený vertex buffer, ktorý užívateľ špecifikuje názvom SSBO bloku (engine si na základe názvu zistí `ID` bufferu) pomocou konfiguračného parametru OpenGL programu. V prípade že užívateľ pomocou konfiguračného parametru OpenGL programu špecifikoval aj index buffer, je vytvorenému VAO priradený pomocou funkcie `glVertexArrayElementBuffer`. V atribúte `varray` triedy `Program` je uchovávané `ID` vytvoreného VAO.

Vytvorenie framebufferu

Vytvorenie framebufferu je založené na návrhu jeho vytvorenia a konfigurácie, popísaného v rámci sekcie 4.4. Aby bol nový framebuffer vytvorený, musí to užívateľ špecifikovať pomocou príslušného konfiguračného parametru vykresľovacieho OpenGL programu.

Ak má byť teda framebuffer vytvorený, je najprv pomocou funkcie `glGetProgramInterfaceiv` zistený počet aktívnych výstupných premenných OpenGL programu (Fragment shaderu). Pre každú výstupnú premennú je zistený jej názov pomocou funkcie `glGetProgramResourceName` a jej index pomocou funkcie `glGetProgramResourceIndex`. Pomocou funkcie `glCreateFramebuffers` je vytvorený OpenGL objekt framebufferu. Jednotlivé výstupné premenné predstavujú objekty textúr. Je teda využitá funkcia `createTexture` triedy `Engine` popísaná v rámci tvorby objektov textúr. Jednotlivé textúry sú na framebuffer naviazané na základe indexu pomocou funkcie `glNamedFramebufferTexture`. V atribúte `framebuffer` triedy `Program` je uchovávané `ID` vytvoreného framebufferu.

5.3 Spúšťanie programov

V návrhu spôsobu spúšťania jednotlivých OpenGL programov, viď sekcia 4.5, je popísaný problém špecifikácie parametrov v rámci spustenia vykreslenia alebo výpočtu. Riešenie tohto problému spočíva vo vytvorení vstavaného SSBO, ktorý bude obsahovať požiadavky na vykreslenie a ďalšieho vstavaného SSBO, ktorý bude obsahovať počet pracovných skupín. Samotné vykreslenie alebo výpočet je typicky možné spustiť následujúcimi funkciami OpenGL:

- `glDispatchCompute(x, y, z)` – Funkcia, ktorá spúšťa výpočet. V rámci parametrov je nutné špecifikovať počet pracovných skupín.
- `glDrawElements(mode, count, type)` – Funkcia, ktorá spúšťa vykreslenie. Grafická pipeline využije „indexované“ načítanie vertexov pomocou index bufferu. Parameter `mode` špecifikuje typ primitíva, parameter `count` špecifikuje počet vertexov a parameter `type` špecifikuje dátový typ prvkov index bufferu.
- `glDrawArrays(mode, first, count)` – Funkcia ktorá spúšťa vykreslenie. Grafická pipeline využije klasické načítanie vertexov pomocou vertex bufferu. Parameter `mode` špecifikuje typ primitíva, parameter `first` špecifikuje index prvého vertexu a parameter `count` špecifikuje počet vertexov.

Jedna možnosť teda bola načítať hodnoty z týchto SSBO pred každým novým snímkom a použiť ich v rámci parametrov jednotlivých volaní. To je ale celkom komplikované vzhľadom na veľký počet variácií spúšťacích funkcií. Existujú ale špeciálne variácie, ktoré nevyžadujú špecifikáciu väčšiny týchto parametrov:

- `glDispatchComputeIndirect` – Počet pracovných skupín je načítaný z bufferu, ktorý je naviazaný na typ `GL_DISPATCH_INDIRECT_BUFFER`. Očakávaný formát dát tvoria 3 premenné typu int.
- `glMultiDrawElementsIndirect` – Požiadavky na vykreslenie sú načítané z bufferu, ktorý je naviazaný na typ `GL_DRAW_INDIRECT_BUFFER`. Očakávaný formát dát tvorí sekvencia štruktúr o 5 premenných typu int.
- `glMultiDrawArraysIndirect` – Požiadavky na vykreslenie sú načítané z bufferu, ktorý je naviazaný na typ `GL_DRAW_INDIRECT_BUFFER`. Očakávaný formát dát tvorí sekvencia štruktúr o 4 premenných typu int.

Vďaka týmto „indirect“ funkciám stačí, aby boli buffery vstavaných SSBO naviazané na potrebné typy. Pokiaľ teda užívateľ upraví dáta týchto SSBO, automaticky sa to odzrkadlí vo volaní týchto funkcií. Na prvý pohľad sa zdá, že funkcie `glMultiDrawElementsIndirect` a `glMultiDrawArraysIndirect` vyžadujú dva odlišné SSBO, vzhľadom na odlišný formát dát. Obidve funkcie našťastie umožňujú špecifikovať dĺžku medzery v pamäti medzi jednotlivými štruktúrami, teda je pre obidve volania možné využiť to isté SSBO. Pri volaní funkcie `glMultiDrawArraysIndirect` je táto medzera nastavená na veľkosť premennej typu int, aby kompenzovala veľkosť štruktúry očakávanej funkciou `glMultiDrawElementsIndirect`. Tento mechanizmus bol inšpirovaný článkom o spôsobe zadávania požiadavkov na vykreslenie zo strany GPU [3].

Engine teda po spracovaní zdrojového kódu aplikácie vytvorí dva buffery pomocou funkcie `glCreateBuffers`. Prvý je naviazaný na typ `GL_DISPATCH_INDIRECT_BUFFER` a bude obsahovať počet pracovných skupín. Druhý je naviazaný na typ `GL_DRAW_INDIRECT_BUFFER` a bude obsahovať požiadavky na vykreslenie. V rámci kompilácie sú do jednotlivých shaderov vložené deklarácie SSBO blokov s vyhradenými binding pointami, na ktoré sú tieto buffery naviazané. Pomocou funkcie `glNamedBufferData` je počet pracovných skupín v prvom bufferi inicializovaný na jednu pracovnú skupinu.

5.4 Nápomocné funkcie

Jazyk GLSL síce disponuje rôznymi vstavanými matematickými funkciami, ale chýbajú mu niektoré základné funkcie, ktoré sú bežne využívané pri práci s 3D grafikou. Jedná sa napríklad o funkcie lineárnej transformácie, prípadne vytvorenie matice perspektívnej projekcie. Z tohto dôvodu engine v rámci mechanizmu vkladania deklarácii do všetkých shaderov vkladá aj tieto funkcie, ktoré môže užívateľ využiť. Jedná sa napríklad o:

- `perspective` – Funkcia, ktorá na základe rôznych parametrov vytvorí maticu perspektívnej projekcie.
- `rotate` – Funkcia, ktorá na vstupnú maticu aplikuje rotačnú maticu.
- `translate` – Funkcia, ktorá na vstupnú maticu aplikuje translačnú maticu.
- `scale` – Funkcia, ktorá na vstupnú maticu aplikuje maticu pre zmenu mierky.
- `rand` – Funkcia generujúca pseudo-náhodné číslo. [9].

5.5 Výsledná štruktúra

Úlohou vstupného bodu programu, funkcie `main`, je vytvoriť objekt triedy `Engine` a zavolať jej metódu `init(filename)`, ktorá je popísaná v nasledujúcej podsekcii 5.5.1. Cesta k súboru je špecifikovaná v prvom parametri príkazovej riadky. Vo while cykle, ktorého podmienka ukončenia je zatvorenie vykresľovacieho okna, je spúšťaná metóda `update` triedy `Engine`, ktorá je popísaná v nasledujúcej podsekcii 5.5.2. Každá iterácia tohto cyklu teda predstavuje jeden snímok aplikácie. Po ukoneční cyklu je zavolaná metóda `destroy` triedy `Engine`, ktorá vyčistí alokované zdroje a engine ukončí.

Pre odchytávanie chýb, ktoré sa môžu vyskytnúť počas behu enginu sú využité `C++` výnimky (exceptions). Obsah funkcie `main` je obalený v try-catch bloku, ktorý výnimku odchytí, obsah výnimky vypíše na štandardný výstup a program riadne ukončí.

5.5.1 Inicializácia

Metóda inicializácie triedy `Engine` na začiatku spracuje konfiguračné parametre aplikácie, viď sekcia 5.1. Pomocou funkcie `glfwCreateWindow` knižnice GLFW následne vytvorí a nastaví okno operačného systému, ktoré predstavuje predvolený framebuffer. Po vytvorení okna inicializuje OpenGL pomocou knižnice GLEW. Nasleduje spracovanie OpenGL programov zo zdrojového kódu, viď sekcia 5.2. Následne sú vytvorené vstavané buffery, viď sekcia 5.3. Inicializácia ešte vytvára jeden vstavaný buffer, engine buffer, popísaný v sekcii 4.6. Podobne ako pri zvyšných vstavaných bufferoch, v rámci kompilácie je do jednotlivých shaderov vložená deklarácia SSBO bloku s vyhradeným binding pointom, na ktorý je engine buffer naviazaný. Pomocou funkcií `glfwSetCursorPosCallback`, `glfwSetKeyCallback`, `glfwSetFramebufferSizeCallback` a `glfwSetMouseButtonCallback` sú naviazané funkcie pre detekciu zmeny stavu myši a klávesnice a aktuálny stav je ukladaný do atribútu `engineBuffer` triedy `Engine`.

5.5.2 Spustenie

Metóda `update` triedy `Engine` má za úlohu postupne spustiť OpenGL programy uložené v atribúte `programs` triedy `Engine`. Ešte pred ich spustením je ale v atribúte `engineBuffer` triedy `Engine` aktualizovaná informácia o čase pomocou funkcie `glfwGetTime`. Obsah tohto atribútu je následne uložený do engine bufferu pomocou funkcie `glNamedBufferData`.

Jednotlivé programy sú spúšťané v rámci for cyklu. Ako prvé je program aktivovaný funkciou `glUseProgram`. Buffery uložené v atribúte `buffers` daného programu sú naviazané na odpovedajúce binding pointy SSBO blokov funkciou `glBindBufferBase`. Textúry uložené v atribúte `textures` daného programu sú naviazané na odpovedajúce uniformné premenné funkciou `glBindImageTexture` v prípade že sa jedná o uniformnú premennú typu image a funkciou `glBindTextureUnit` v prípade že sa jedná o uniformnú premennú typu sampler. Pokiaľ sa jedná o vykresľovací OpenGL program, je navyše aktivovaný framebuffer a VAO daného programu. Výpočtový OpenGL program je spustený funkciou `glDispatchComputeIndirect`. Vykresľovací OpenGL program je spustený funkciou `glMultiDrawElementsIndirect` v prípade že užívateľ špecifikoval index buffer pomocou konfiguračnej premennej OpenGL programu. V opačnom prípade je vykresľovací OpenGL program spustený funkciou `glMultiDrawArraysIndirect`. V prípade že užívateľ špecifikoval, aby sa OpenGL spustil iba raz pomocou konfiguračného parametru, je nastavený atribút `isIgnored` triedy `Program` a v ďalších iteráciách bude tento program na základe tohto atribútu ignorovaný.

Kapitola 6

Dosiahnuté výsledky

Hlavným cieľom vytvoreného enginu je urýchliť tvorbu a prototypovanie OpenGL programov a shaderov. Tento cieľ by mal byť zaručený tým, že engine automatizuje konfiguračnú časť a užívateľ sa môže sústrediť na implementáciu samotného problému. Úlohou tejto kapitoly je ukázať, že bol tento cieľ úspešne dosiahnutý.

Za účelom porovnania výsledkov bola vymyslená jednoduchá[1] úloha: Vykresliť trojuholník pomocou GPU, ktorého farba sa mení v závislosti od stlačenej klávesy. Táto úloha bola implementovaná dvoma rôznymi spôsobmi: manuálne pomocou OpenGL API a pomocou vytvoreného enginu. Jednotlivé implementácie neboli zbytočne naťahované, aby bolo porovnanie čo najpresnejšie.

Prvá implementácia bola teda vytvorená v jazyku C++ s využitím knižníc GLFW a GLEW, ktoré boli využité aj pri implementácii enginu. Spracovanie stlačených kláves a následná aktualizácia farby sa vykonáva na strane CPU. Aktuálna farba je odosielaná pomocou uniformnej premennej jedinému OpenGL programu, ktorý tento trojuholník vykreslí. Súčasťou implementácie je riadne ošetrenie chybových stavov a vyčistenie alokovaných zdrojov po ukončení. Ošetrenie chybových stavov nemusí byť pri triviálnych úlohách podstatné, ale pri komplexnejších úlohách chce mať užívateľ typicky informáciu o tom čo zlyhalo, teda je vhodné ho do implementácie zahrnúť.

Druhá implementácia bola vytvorená pomocou enginu. Keďže limitáciou enginu je nemožnosť použitia uniformných premenných pre zasielanie vlastných dát, bolo nutné použiť SSBO, ktoré môžu byť obecne pomalšie (keďže podporujú aj zápis nie len čítanie). Implementácia teda využíva SSBO, v ktorom je uložená aktuálna hodnota farby a výpočtový program, ktorý na základe stlačenej klávesy upravuje túto hodnotu rovnakým spôsobom ako prvá implementácia.

Pre riešenie tejto konkrétnej úlohy nie je SSBO na ukladanie aktuálnej farby úplne ideálne a uniformná premenná je vhodnejšia. Užívateľ ale môže dostať inú úlohu, pre ktorej riešenie je optimálne práve SSBO. Z tohto dôvodu bola vytvorená ešte tretia implementácia, ktorá je vytvorená rovnakým spôsobom ako prvá ale s využitím SSBO. Táto implementácia by mala zvýrazniť reálny rozdiel, keďže sa snaží implementovať úlohu rovnako (ako pomocou enginu) a nerieši či je implementovaná optimálne.

Na tabuľke 6.1 je možné vidieť rozdiel v počte riadkov zdrojového kódu medzi jednotlivými implementáciami. Už pri takto jednoduchej úlohe je rozdiel medzi manuálnou implementáciou a implementáciou pomocou enginu takmer dvojnásobný. Dôležitý je aj pomer

[1] Úloha nemusí byť zložitá. Pokiaľ engine ušetrí čas už pri triviálnych scénach, o to viac ho ušetrí pri komplexnejších scénach, keďže budú využívať väčší počet objektov, ktoré treba nakonfigurovať.

celkového počtu riadkov a počtu riadkov v shaderoch, ktorý do istej miery ukazuje v akom pomere užívateľ konfiguroval a implementoval. Kód samotných shaderov je pri manuálnej implementácii nižší. Tento rozdiel je spôsobený tým, že v implementácii pomocou enginu je celá logika vykonávaná shaderami, vrátane inicializácie bufferov a spracovania stlačených kláves. Samotný rozdiel a pomer teda lepšie zvýrazňuje manuálna implementácia pomocou SSBO, kde spracovanie stlačených kláves prebieha tiež v rámci shaderu.

	Manuálne	Engine	Manuálne (SSBO)
Celkový počet riadkov	94	52	136
Počet riadkov v Shaderoch	13	29	31
Priemerná doba spracovania snímku [μs]	140	56	151

Tabuľka 6.1: Porovnanie celkového počtu riadkov zdrojového kódu a z toho počtu riadkov shaderov potrebných pre implementáciu danej úlohy. V jednotlivých počtoch nie sú zarátané prázdne riadky, komentáre a príkazy include. Tabuľka ďalej obsahuje porovnanie priemernej doby spracovania jedného snímku v mikrosekundách na zariadení s CPU AMD Ryzen 5 1600 a GPU NVIDIA GeForce GTX 1050 Ti.

Konfigurácia OpenGL objektov je zdĺhavá a relatívne náročná, čo už bolo vidno pri implementácii enginu. Značný rozdiel v počte riadkov sa dal teda predpokladať aj bez toho aby bol tento test vytvorený, keďže engine tieto objekty nakonfiguruje za užívateľa automaticky. Veľmi dôležité je ale podotknúť aj na fakt, že pri tvorbe aplikácie častokrát programátor nevie, ako daný problém implementovať. Časté zmeny kódu shaderu znamenajú to, že je nutné upraviť aj konfiguráciu samotných objektov, čo v konečnom dôsledku môže znamenať navýšenie pomeru konfigurácie ku implementácii.

Rozdiel priemernej doby spracovania snímku ukazuje, že v svojej podstate nie je engine nutne pomalší. Pri implementácii tejto konkrétnej úlohy je dokonca rýchlejší. Dôvodom je to, že manuálne implementácie využívajú priame volanie spustenia vykresľovania, zatiaľ čo engine využíva nepriame volanie (indirect), ktoré je obecne rýchlejšie, keďže pracuje priamo s VRAM. Aby bola teda manuálna implementácia rovnako rýchla, stačilo by toto nepriame volanie využiť aj v nej, čo by ale znamenalo dodatočnú konfiguráciu. Naopak z dôvodu že v rámci enginu nie je možné prispôsobiť každý parameter každého objektu alebo funkcie, môže veľmi ľahko nastať situácia kedy bude engine pomalší. Manuálne implementácia je teda síce oveľa flexibilnejšia, ale na úkor celkovej rýchlosti implementácie v závislosti od skúseností a vedomostí užívateľa.

Kapitola 7

Záver

V rámci tejto práce bol úspešne vytvorený nástroj, ktorý umožňuje jednoduchú tvorbu a prototypovanie OpenGL programov. Tento nástroj na základe zdrojového kódu aplikácie automaticky vytvorí a nakonfiguruje základné objekty potrebné pre spustenie aplikácie. V rámci zdrojového kódu je možné definovať sekvenciu ľubovoľných OpenGL programov a rôzne konfiguračné parametre, ktoré prispôsobujú chovanie nástroja. Užívateľ sa vďaka vytvorenému nástroju vyhne častokrát repetetívnej a zdĺhavej manuálnej konfigurácii a môže sa venovať samotnej implementácii. Vďaka nástroju je taktiež možné pracovať s myšou a klávesnicou priamo na GPU. Pre overenie funkčnosti enginu bolo vytvorených viacero demonštračných aplikácií. Taktiež bolo vytvorené krátke demonštračné video[1].

Napriek tomu, že bol hlavný cieľ tejto práce splnený, je nutné poukázať na limitácie vytvoreného nástroja. Keďže je API OpenGL relatívne obsiahla, je ťažké pokryť každú jednu dostupnú funkciu a umožniť jej automatickú konfiguráciu. Najväčšia limitácia vytvoreného nástroja je podpora načítavania externých zdrojov, ako napríklad textúr a 3D modelov zo súborového systému. Podporovaná je len procedurálna tvorba, kde si užívateľ musí textúry a modely vygenerovať sám v rámci aplikácie. Potencionálna budúca práca by teda spočívala v podpore načítavania externých zdrojov a v rozšírení už existujúcej funkcionality. Jedná sa napríklad o podporu ďalších periférií a možnosti hlbšej konfigurácie parametrov bufferov, textúr a framebufferov, keďže v aktuálnej implementácii je podporovaná iba ich základná konfigurácia. Jeden z ambicióznejších plánov by bola podpora užívateľského rozhrania, plne vykresľovaného a ovládaného z prostredia shaderov a pomocné funkcie, ktoré by uľahčovali paralelné spracovanie.

V konečnom dôsledku je vytvorený nástroj použiteľný a plne dostačujúci pre tvorbu jednoduchších grafických aplikácií. Pre tvorbu komplexnejších aplikácií vyžadujúcich detailnejšiu konfiguráciu je ale potrebné nástroj rozšíriť vyššie uvedenými návrhmi.

[1] Demonštračné video dostupné online: https://youtu.be/iW24gOjQeSQ

Literatúra

[1] Gray, A. *GPU Architecture* [online]. ARCHER, november 2017 [cit. 2022-21-04]. Dostupné z: https://www.archer.ac.uk/training/course-material/2017/11/gpu-daresbury/slides/GPU_Architecture.pdf.

[2] Kessenich, J., Baldwin, D. a Rost, R. *The OpenGL Shading Language* [online]. Khronos Group, 10. júla 2019 [cit. 2022-21-04]. Dostupné z: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

[3] Lingtorp, A. *Generating draw commands on the GPU in OpenGL* [online], 05. decembra 2018 [cit. 2022-21-04]. Dostupné z: https://lingtorp.com/2018/12/05/OpenGL-SSBO-indirect-drawing.html.

[4] Lou, Y. *Introduction to C - Preprocessor* [online], 02. septembra 2011 [cit. 2022-01-05]. Dostupné z: http://www.cs.cornell.edu/courses/cs2022/2011sp/lectures/lect08.pdf.

[5] McReynolds, T. a Blythe, D. *Advanced Graphics Programming Using OpenGL*. Elsevier, 2005. ISBN 1-55860-659-9. Dostupné z: http://www.r-5.org/files/books/computers/algo-list/realtime-3d/Tom_McReynolds-Advanced_Graphics_Programming-EN.pdf.

[6] Micikevicius, P. *Performance Optimization: Programming Guidelines and GPU Architecture Reasons Behind Them* [online]. GPU Technology Conference, 2013 [cit. 2022-21-04]. Dostupné z: https://on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf.

[7] Rege, A. *An Introduction to Modern GPU Architecture* [online]. NVIDIA [cit. 2022-21-04]. Dostupné z: http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf.

[8] Segal, M. a Akeley, K. *The OpenGL Graphics System: A Specification* [online]. Khronos Group, 22. októbra 2019 [cit. 2022-21-04]. Dostupné z: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf.

[9] Vivo, P. G. a Lowe, J. *The Book of Shaders: Random* [online]. 2015 [cit. 2022-01-05]. Dostupné z: https://thebookofshaders.com/10/.

[10] Vries, J. de. *Learn OpenGL - Graphics Programming*. Kendall & Welling, 2020. ISBN 978-90-90-33256-7. Dostupné z: https://learnopengl.com/book/book_pdf.pdf.